МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
ФАКУЛЬТЕТ ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

Кафедра интеллектуальных информационных технологий

Отчёт по лабораторной работе №3

Специальность ПО11

Выполнил
И. А. Гурин
студент группы ПО11

Проверил
А. А. Крощенко
ст. преп. кафедры ИИТ,
15.03.2025 г.

Брест 2025

Цель работы: приобрести навыки применения паттернов проектирования при решении практических задач с использованием языка Python

Общее задание

• Прочитать задания, взятые из каждой группы, соответствующей одному из трех основных типов паттернов;

• Определить паттерн проектирования, который может использоваться при реализации задания. Пояснить свой выбор;

• Реализовать фрагмент программной системы, используя выбранный паттерн. Реализовать все необходимые дополнительные классы.

Задание 1. Торговый автомат с возможностью выдачи любого выбранного товара (шоколадные батончики, чипсы, пакетированные соки и т.д.)

При реализиции задания можно использовать паттерн проектирования фабрика т. к. нам необходимо отделить логику создания объектов от их использования и упростить добавление новых продуктов. Также необходимо сокрыть детали реализации

Выполнение:

Код программы:

```
class Product():
    def __init__(self, name):
        self.name = name

    def getInfo(self):
        return f"Product: {self.name}"

class ChocolateBar(Product):
    def __init__(self):
        super().__init__("ChocolateBar")

class Chips(Product):
    def __init__(self):
        super().__init__("Chips")

class Juice(Product):
    def __init__(self):
        super().__init__("Juice")

class ProductFactory:
    products = {
        "ChocolateBar": ChocolateBar,
        "Chips": Chips,
        "Juice": Juice,
    }

    @staticmethod
    def createProduct(product_name: str) -> Product:
        product_class = ProductFactory.products.get(product_name)
        if product_class:
            return product_class()
        raise ValueError("Unknown type of products")

if __name__ == "__main__":
    while True:
        product_type: str = input("Enter the type of product: ")
        if product_type == "q": break
        try:
            product: Product = ProductFactory.createProduct(product_type)
            print(product.getInfo() + ' created!   ')
        except ValueError as e:
            print(e)
```

Рисунки с результатами работы программы:

```
Enter the type of product: Juice
Product: Juice created!
Enter the type of product: Chips
Product: Chips created!
Enter the type of product: Juice
Product: Juice created!
Enter the type of product: q
```

Задание 2. ДУ телевизора. Реализовать иерархию телевизоров для конкретных производителей и иерархию средств дистанционного управления. Телевизоры должны иметь присущие им атрибуты и функции. ДУ имеет набор функций для изменения текущего канала, увеличения/уменьшения громкости, включения/выключения телевизора и т.д. Эти функции должны отличаться для различных устройств ДУ.

При реализиции задания можно использовать паттерн проектирования мост. Он используется для разделения абстракции (например, класса пульта дистанционного управления) и его реализации (устройств, с которыми он работает) таким образом, чтобы они могли изменяться независимо друг от друга. Это позволяет избежать жёсткой зависимости между этими компонентами.

Выполнение:

Код программы:

```python
from abc import ABC, abstractmethod
from typing import override

class RemoteControl(ABC):
    def __init__(self, device):
        self.device = device

    @abstractmethod
    def turn_on(self):
        pass

    @abstractmethod
    def turn_off(self):
        pass

    @abstractmethod
    def volume_up(self):
        pass

    @abstractmethod
    def volume_down(self):
        pass

    @abstractmethod
    def change_channel(self, channel):
        pass

class BasicRemote(RemoteControl):
    def turn_on(self):
        print("Базовое ДУ: Включение телевизора.")
        self.device.power_on()

    def turn_off(self):
        print("Базовое ДУ: Выключение телевизора.")
        self.device.power_off()

    def volume_up(self):
        print("Базовое ДУ: Увеличение громкости.")
        self.device.increase_volume()
```

```
    def volume_down(self):
        print("Базовое ДУ: Уменьшение громкости.")
        self.device.decrease_volume()

    def change_channel(self, channel):
        print(f"Базовое ДУ: Смена канала на {channel}.")
        self.device.set_channel(channel)

class AdvancedRemote(BasicRemote):
    @override
    def turn_on(self):
        print("Продвинутое ДУ: Включение телевизора.")
        self.device.power_on()

    @override
    def turn_off(self):
        print("Продвинутое ДУ: Выключение телевизора.")
        self.device.power_off()

    @override
    def volume_up(self):
        print("Продвинутое ДУ: Увеличение громкости.")
        self.device.increase_volume()

    @override
    def volume_down(self):
        print("Продвинутое ДУ: Уменьшение громкости.")
        self.device.decrease_volume()

    @override
    def change_channel(self, channel):
        print(f"Продвинутое ДУ: Смена канала на {channel}.")
        self.device.set_channel(channel)

    def mute(self):
        print("Продвинутое ДУ: Отключение звука.")
        self.device.mute()

class TV(ABC):
    def __init__(self, brand):
        self.brand = brand
        self.is_on = False
        self.volume = 10
        self.channel = 1

    def power_on(self):
        self.is_on = True
        print(f"{self.brand}: Телевизор включен.")

    def power_off(self):
        self.is_on = False
        print(f"{self.brand}: Телевизор выключен.")

    def increase_volume(self):
        if self.is_on:
            self.volume += 1
            print(f"{self.brand}: Громкость увеличена до {self.volume}.")

    def decrease_volume(self):
        if self.is_on and self.volume > 0:
            self.volume -= 1
            print(f"{self.brand}: Громкость уменьшена до {self.volume}.")

    def mute(self):
        if self.is_on and self.volume > 0:
            self.volume = 0
            print(f"{self.brand}: Громкость уменьшена до {self.volume}.")

    def set_channel(self, channel):
        if self.is_on:
            self.channel = channel
```

```
            print(f"{self.brand}: Канал переключен на {self.channel}.")

class SamsungTV(TV):
    def __init__(self):
        super().__init__("Samsung")

class SonyTV(TV):
    def __init__(self):
        super().__init__("Sony")

def main():
    samsung_tv = SamsungTV()
    sony_tv = SonyTV()

    basic_remote = BasicRemote(samsung_tv)
    advanced_remote = AdvancedRemote(sony_tv)

    basic_remote.turn_on()
    basic_remote.volume_up()
    basic_remote.change_channel(5)
    basic_remote.turn_off()

    print()

    advanced_remote.turn_on()
    advanced_remote.mute()
    advanced_remote.change_channel(10)
    advanced_remote.turn_off()

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

Рисунки с результатами работы программы:

```
Базовое ДУ: Включение телевизора.
Samsung: Телевизор включен.
Базовое ДУ: Увеличение громкости.
Samsung: Громкость увеличена до 11.
Базовое ДУ: Смена канала на 5.
Samsung: Канал переключен на 5.
Базовое ДУ: Выключение телевизора.
Samsung: Телевизор выключен.

Продвинутое ДУ: Включение телевизора.
Sony: Телевизор включен.
Продвинутое ДУ: Отключение звука.
Sony: Громкость уменьшена до 0.
Продвинутое ДУ: Смена канала на 10.
Sony: Канал переключен на 10.
Продвинутое ДУ: Выключение телевизора.
Sony: Телевизор выключен.
```

Задание 3. Вспомогательная библиотека для работы с текстовыми файлами. Должны быть предусмотрены следующие функции: чтение одного или нескольких файлов, изменение файла(ов), отмена последней выполненной операции (одной в случае простой единичной операции или нескольких в случае сложной операции), последовательное выполнение нескольких операций.

При реализиции задания можно использовать паттерн проектирования команда. Он используется для инкапсуляции запроса на выполнение операции в отдельный объект, который содержит всю необходимую информацию для выполнения действия, его отмены или повторного выполнения. Мы его используем потому что у каждого объекта команды (Command) есть метод undo, который

позволяет откатывать изменения. Добавление новых операций (команд) легко реализуется через создание новых классов, наследующих абстрактный класс Command. Паттерн чётко отделяет логику выполнения команды от её использования. FileManager отвечает за выполнение и историю команд, а сами команды отвечают за реализацию специфических действий.

Код программы:

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def undo(self):
        pass

class ReadFileCommand(Command):
    def __init__(self, filepath):
        self.filepath = filepath
        self.content = None

    def execute(self):
        with open(self.filepath, 'r') as file:
            self.content = file.read()
        print(f"Прочитан файл: {self.filepath}")
        return self.content

    def undo(self):
        print("Операция чтения файла не требует отмены.")

class ReadMultipleFilesCommand(Command):
    def __init__(self, filepaths):
        self.filepaths = filepaths
        self.contents = {}

    def execute(self):
        for filepath in self.filepaths:
            try:
                with open(filepath, 'r') as file:
                    self.contents[filepath] = file.read()
                print(f"Прочитан файл: {filepath}")
            except FileNotFoundError:
                print(f"Файл {filepath} не найден.")
        return self.contents

    def undo(self):
        print("Операция чтения файлов не требует отмены.")

class WriteFileCommand(Command):
    def __init__(self, filepath, content):
        self.filepath = filepath
        self.content = content
        self.previous_content = None

    def execute(self):
        with open(self.filepath, 'r') as file:
            self.previous_content = file.read()
        with open(self.filepath, 'w') as file:
            file.write(self.content)
        print(f"Файл {self.filepath} изменен.")

    def undo(self):
        with open(self.filepath, 'w') as file:
            file.write(self.previous_content)
        print(f"Изменения в файле {self.filepath} отменены.")

class FileManager:
    def __init__(self):
        self.history = []
```

```
    def execute_command(self, command):
        result = command.execute()
        self.history.append(command)
        return result

    def undo_last_command(self):
        if self.history:
            command = self.history.pop()
            command.undo()
        else:
            print("Нет операций для отмены.")

def main():
    manager = FileManager()

    read_command = ReadFileCommand("example.txt")
    content = manager.execute_command(read_command)
    print(content)

    read_multiple_files_command = ReadMultipleFilesCommand(["example.txt", "example2.txt"])
    contents = manager.execute_command(read_multiple_files_command)
    for filepath, content in contents.items():
        print(f"Содержимое файла {filepath}:")
        print(content)

    write_command = WriteFileCommand("example.txt", "Новое содержимое файла.")
    manager.execute_command(write_command)

    content = manager.execute_command(read_command)
    print(content)

    manager.undo_last_command()

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

Рисунки с результатами работы программы:

```
Прочитан файл: example.txt
Новое содержимое файла.
Прочитан файл: example.txt
Прочитан файл: example2.txt
Содержимое файла example.txt:
Новое содержимое файла.
Содержимое файла example2.txt:
Hello world
Файл example.txt изменен.
Прочитан файл: example.txt
Новое содержимое файла.
Операция чтения файла не требует отмены.
```

Вывод: приобрёл навыки применения паттернов проектирования при решении практических задач с использованием языка Python