

### ПЗ 03.01 Работа с поведенческими паттернами

**Задание:** Реализовать проекты по предметной области в соответствии с вариантом.

1 проект. (Шаблонный метод)

**Задание:** Создание консольного приложения для обработки текстовых файлов шаблонами.

1) Приложение предназначено для обработки текстовых файлов с использованием шаблонного метода. Основная задача приложения – выполнение различных этапов предобработки текста, таких как:

1. Очистка от комментариев (Строки начинающиеся с //)
2. Замена переносов строк
3. Удаление лишних пробелов
4. Возможность добавления дальнейших обработок текста

Основные требования:

1. Приложение должно обрабатывать текстовые файлы, применяя последовательность шагов предобработки.
2. Каждый этап обработки должен быть реализован как отдельный шаг.
3. Приложение должно поддерживать дальнейшее расширение без значительного изменения старого кода.

1) UML диаграмма

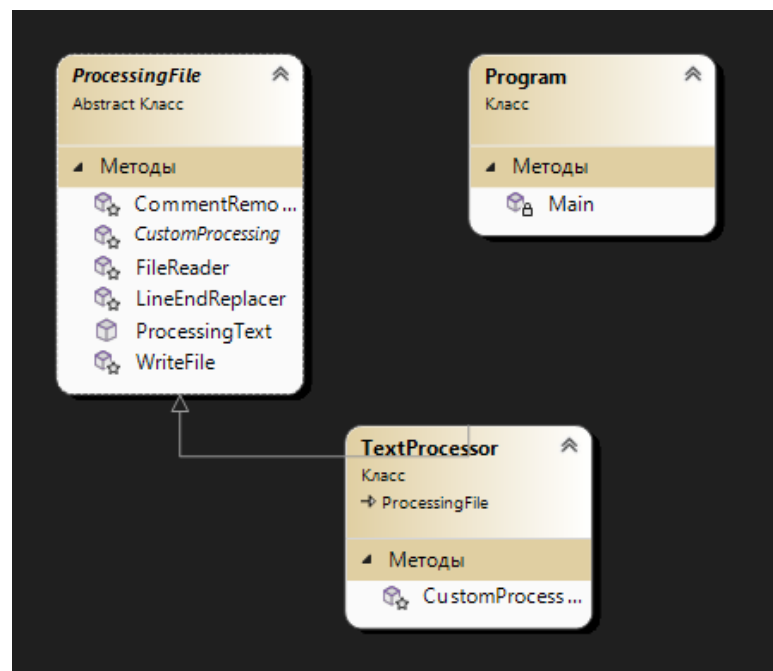


Рисунок 1 – UML диаграмма текстового редактора.

2) Краткий вывод по использованию паттерна в текущей задаче

В проекте текстового редактора использовался шаблонный метод, который позволил разделить и типизировать задачи по предобработке текстовых файлов, позволяя в дальнейшем без изменений старого кода добавлять новые виды обработки текста.

### 3) Листинг кода

```
Ссылка: 0
internal class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        // Имена входного и выходного файлов
        string InputFilePath = "input.txt";
        string OutputFilePath = "output.txt";

        ProcessingFile processing = new TextProcessor();
        processing.ProcessingText(InputFilePath, OutputFilePath);

        Console.WriteLine($"Обработка файла завершена. Результат записан в {OutputFilePath}");
    }
}
```

Рисунок 2 – Основной метод текстового редактора.

В данном методе указанном на рисунке 2, представлен пример использования текстового редактора, пользователь создает 2 строки с исходным названием файла и итоговым, сам файл с текстом должен находиться в папке с исполняемым файлом, после чего пользователь вызывает метод обработки текста и передает в него 2 значения, после чего готовый файл помещается в ту же папку в которой находился исходный файл.

```
abstract class ProcessingFile
{
    // Основной метод
    Ссылка: 1
    public void ProcessingText(string InputFilePath, string OutputFilePath)
    {
        string text = FileReader(InputFilePath);
        text = CommentRemover(text);
        text = LineEndReplacer(text);
        text = CustomProcessing(text);
        WriteFile(OutputFilePath, text);
    }

    // Метод на чтение файла
    Ссылка: 1
    protected string FileReader(string filePath)
    {
        return File.ReadAllText(filePath, Encoding.UTF8);
    }

    // Метод на запись в файл
    Ссылка: 1
    protected void WriteFile(string filePath, string text)
    {
        File.WriteAllText(filePath, text, Encoding.UTF8);
    }

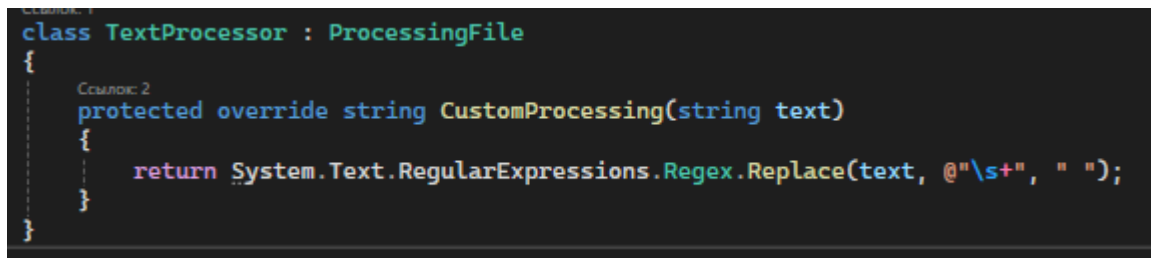
    // Метод заменяющий комментарии
    Ссылка: 1
    protected string CommentRemover(string text)
    {
        return System.Text.RegularExpressions.Regex.Replace(text, @"//.*", "");
    }

    // Метод заменяющий переносы
    Ссылка: 1
    protected string LineEndReplacer(string text)
    {
        return text.Replace(Environment.NewLine, " ");
    }

    // Заготовка на дальнейшую обработку
    Ссылка: 2
    protected abstract string CustomProcessing(string text);
}
```

Рисунок 3 – Основная логика текстового редактора.

На рисунке 3 представлена основной логический класс, выполняющий основную работу приложения, в нем расположены методы предобработки, которые шаг за шагом редактируют текст, а так же методы чтения исходного и записи итогового текста в файл по средствам System.IO, предоставляющую возможность использовать встроенный метод побайтового чтения и записи. Так же данный класс является абстрактным и его можно будет использовать в будущем как каркас под новые шаги обработки текста, пример этого способа представлен на рисунке 4.



```
class TextProcessor : ProcessingFile
{
    Ссылка: 2
    protected override string CustomProcessing(string text)
    {
        return System.Text.RegularExpressions.Regex.Replace(text, @"\s+", " ");
    }
}
```

Рисунок 4 – Пример добавления шагов обработки.

## 2 проект (Состояние)

1) Задание: Создать консольную программу для просмотра и изменения курсов валют в зависимости от состояния.

Программа предназначена для отображения курсов валют и управления режимами работы: Режим просмотра и режим обновления данных. В зависимости от активного режима, программа будет менять свое поведение:

- В режиме просмотра программа отображает текущие курсы валют.
- В режиме обновления программа обновляет курсы валют и переключается в режим просмотра.

### Основные требования:

1. Отображения курсов валют.
2. Обновление курсов валют.
3. Управление режимами.

## 2) UML диаграмма классов



Рисунок 5 – UML диаграмма программы курса валют

### 3) Краткий вывод по использованию паттерна в текущей задаче

В программе отображения курса валют, паттерн состояния играл основную роль, меняя поведения программы в соответствии со своим значением, благодаря чему программа ограничивала некоторые действия, выполнение которых недопустимо в текущем состоянии.

### 4) Листинг кода

```

internal class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        var context = new ExchangeRateContext();

        while (true)
        {
            Console.WriteLine("\nВыберите действие:");
            Console.WriteLine("1 - Показать курсы валют");
            Console.WriteLine("2 - Обновить курсы валют");
            Console.WriteLine("3 - Выйти");
            Console.Write("Ваш выбор: ");

            var sw = Console.ReadLine();

            switch (sw)
            {
                case "1":
                    context.ShowExchangeRates();
                    break;
                case "2":
                    context.UpdateExchangeRates();
                    break;
                case "3":
                    return;
                default:
                    Console.WriteLine("Неверный выбор. Попробуйте снова.");
                    break;
            }
        }
    }
}

```

Рисунок 6 – Основной метод управления программой курса валют

На рисунке 6 изображен основной метод управления программой, он состоит из небольшого визуального сопровождения для удобства использования программы а так же switch который и вызывает тот или иной метод по команде пользователя.

```
Ссылка: 3
class ViewMode : IState
{
    private readonly ExchangeRateContext _context;
    Ссылка: 2
    public ViewMode(ExchangeRateContext context) => _context = context;

    Ссылка: 2
    public void ShowExchangeRates()
    {
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Текущие курсы валют:");
        foreach (var rate in _context.GetSavedRates())
        {
            Console.WriteLine($"{rate.Key}: {rate.Value}");
        }
        Console.ForegroundColor = ConsoleColor.White;
    }

    Ссылка: 2
    public void UpdateExchangeRates()
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Переключение в режим обновления данных...");
        _context.SetState(new UpdateMode(_context));
        Console.ForegroundColor = ConsoleColor.White;
    }
}
```

Рисунок 7 – Режим просмотра курсов.

На рисунке 7 представлен класс выполняющий режим просмотра курсов валют, в нем представлены 2 метода, отображение текущих курсов валют, которые хранятся в справочнике валют, а так же метод обновления курсов валют, выводящий сообщение о переключении в режим обновления данных и переключающий состояние в режим обновления.

```
Ссылка: 2
class UpdateMode : IState
{
    private readonly ExchangeRateContext _context;

    Ссылка: 1
    public UpdateMode(ExchangeRateContext context) => _context = context;

    Ссылка: 2
    public void ShowExchangeRates() => Console.WriteLine("Режим обновления данных. Просмотр курсов недоступен.");

    // Вызывать 2 раза
    Ссылка: 2
    public void UpdateExchangeRates()
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.WriteLine("Обновление курсов валют...");
        _context.UpdateRates();
        Console.WriteLine("Курсы обновлены. Переключение в режим просмотра.");
        _context.SetState(new ViewMode(_context));
        Console.ForegroundColor = ConsoleColor.White;
    }
}
```

Рисунок 8 – Режим обновления данных.

На 8 рисунке уже изображен класс обновления данных, который все так же принимает текущее состояние и отвечает за изменение поведения программы на обновление курсов валют.

```
class ExchangeRateContext
{
    private IState _currentState;
    // Словарь валют
    private Dictionary<string, decimal> _exchangeRates;

    Ссылка: 1
    public ExchangeRateContext()
    {
        _currentState = new ViewMode(this);
        _exchangeRates = new Dictionary<string, decimal>
        {
            { "USD", 75.0m},
            { "EUR", 85.0m},
            { "GBP", 95.0m}
        };
    }

    public void SetState(IState newState) => _currentState = newState;

    public void ShowExchangeRates() => _currentState.ShowExchangeRates();

    public void UpdateExchangeRates() => _currentState.UpdateExchangeRates();

    Ссылка: 1
    public Dictionary<string, decimal> GetSavedRates()
    {
        return _exchangeRates;
    }

    // Попытка сделать обновлению валют
    Ссылка: 1
    public void UpdateRates()
    {
        var random = new Random();
        _exchangeRates["USD"] = Math.Round(70 + (decimal)random.NextDouble() * 10, 2);
        _exchangeRates["EUR"] = Math.Round(70 + (decimal)random.NextDouble() * 10, 2);
        _exchangeRates["GBP"] = Math.Round(70 + (decimal)random.NextDouble() * 10, 2);
    }
}
```

Рисунок 9 – Класс изменения курсов валют.

9 рисунок представляет класс, отвечающий за изменение и хранение курсов валют, курсы хранятся в справочнике, состоящем из названия и цены за единицу валюты в рублях, а метод обновления курсов, отвечает за случайное изменение курса валют по средствам Random. Данный класс все так же принимает значение текущего состояния и действует согласно ему.

```
interface IState
{
    void ShowExchangeRates();
    Ссылка: 3
    void UpdateExchangeRates();
}
```

Рисунок 10 – Интерфейс состояния.

### 3 проект (Команды)

Задание: Создать приложение для хранения контактов, которое позволяет добавлять, редактировать и удалять контакты с помощью команд, а также поддерживает отмену этих действий.

1) Приложение предназначено для управления списком контактов. Оно позволяет пользователю выполнять следующие действия:

- Добавлять новые контакты.
- Редактировать существующие контакты.
- Удалять контакты.
- Просматривать список всех контактов.
- Отменять последнее выполненное действие.

2) UML диаграмма классов реализованного проекта

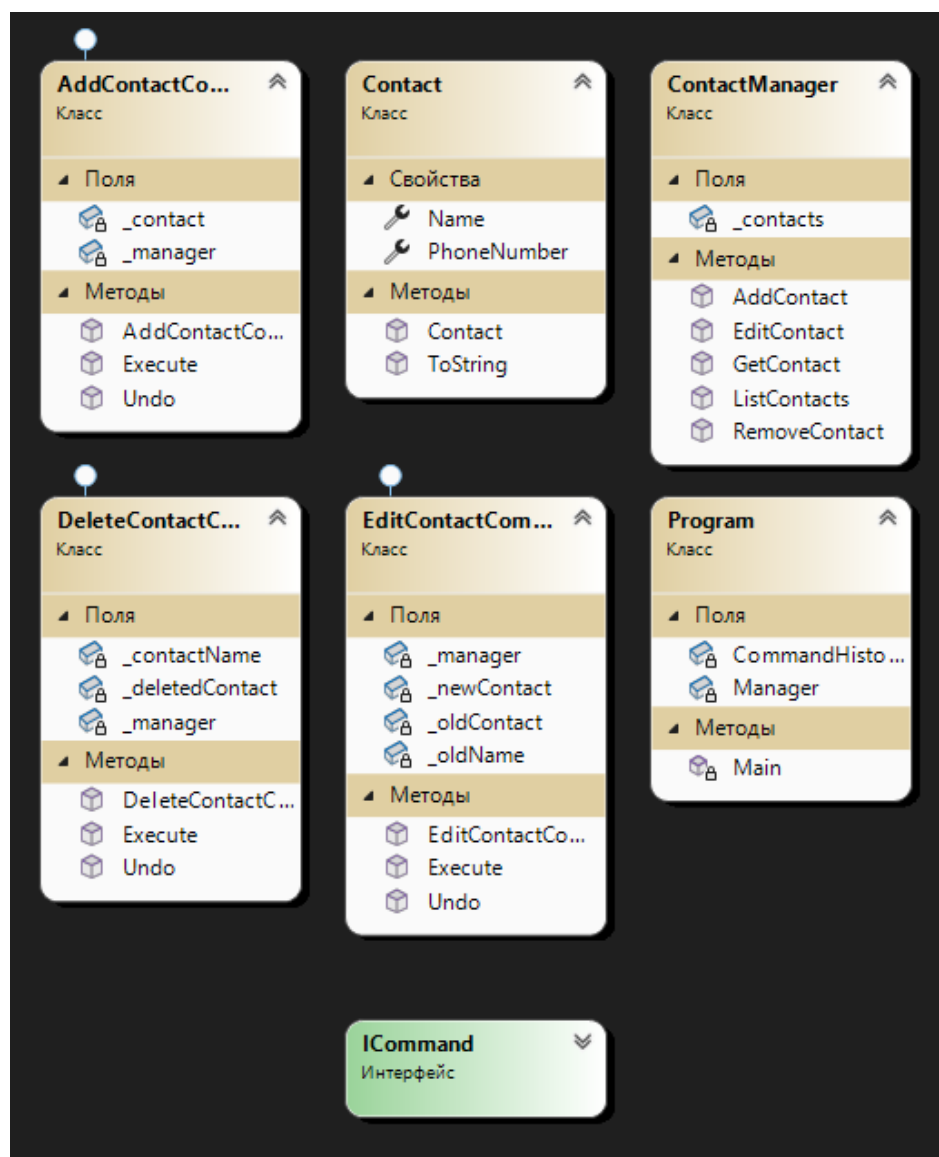


Рисунок 11 – UML диаграмма классов.

3) Краткий вывод по использованию паттерна в текущей задаче

Паттерн команда позволил воссоздать именно тот функционал, который был задан, благодаря нему, в программе появилась возможность использовать методы как объекты и хранение их в стеке для последующей отмены.

#### 4) Листинг кода

```
internal class Program
{
    private static readonly ContactManager Manager = new ContactManager();
    // Для отмены команды используется стек в котором хранятся последние команды
    private static readonly Stack<ICommand> CommandHistory = new Stack<ICommand>();
    static void Main(string[] args)
    {
        while (true)
        {
            Console.WriteLine("\nВыберите действие:");
            Console.WriteLine("1 - Добавить контакт");
            Console.WriteLine("2 - Редактировать контакт");
            Console.WriteLine("3 - Удалить контакт");
            Console.WriteLine("4 - Показать все контакты");
            Console.WriteLine("5 - Отменить последнее действие");
            Console.WriteLine("6 - Выйти");
            Console.Write("Ваш выбор: ");

            var sw = Console.ReadLine();

            switch (sw)
            {
                case "1":
                    AddContact();
                    break;
                case "2":
                    EditContact();
                    break;
                case "3":
                    DeleteContact();
                    break;
                case "4":
                    Manager.ListContacts();
                    break;
                case "5":
                    UndoLastCommand();
                    break;
                case "6":
                    return;
                default:
                    Console.WriteLine("Неверный выбор. Попробуйте снова.");
                    break;
            }
        }
    }
}
```

Рисунок 12 – Вызов методов и управление



```

// Методы
static void AddContact()
{
    Console.WriteLine("Введите имя: ");
    var name = Console.ReadLine();
    Console.WriteLine("Введите номер телефона: ");
    var phoneNumber = Console.ReadLine();

    var contact = new Contact(name, phoneNumber);
    var command = new AddContactCommand(Manager, contact);
    command.Execute();
    CommandHistory.Push(command);
}

static void EditContact()
{
    Console.WriteLine("Введите имя контакта для редактирования: ");
    var oldName = Console.ReadLine();
    Console.WriteLine("Введите новое имя: ");
    var newName = Console.ReadLine();
    Console.WriteLine("Введите новый номер телефона: ");
    var newPhoneNumber = Console.ReadLine();

    var newContact = new Contact(newName, newPhoneNumber);
    var command = new EditContactCommand(Manager, oldName, newContact);
    command.Execute();
    CommandHistory.Push(command);
}

static void DeleteContact()
{
    Console.WriteLine("Введите имя контакта для удаления: ");
    var name = Console.ReadLine();

    var command = new DeleteContactCommand(Manager, name);
    command.Execute();
    CommandHistory.Push(command);
}

static void UndoLastCommand()
{
    if (CommandHistory.Count > 0)
    {
        var command = CommandHistory.Pop();
        command.Undo();
        Console.WriteLine("Последнее действие отменено.");
    }
    else
    {
        Console.WriteLine("Нет действий для отмены.");
    }
}

```

Рисунок 13 – Исполнительные методы

```

interface ICommand
{
    void Execute();
    void Undo();
}

```

Рисунок 14 – Интерфейс

```

class Contact
{
    public string Name { get; set; }
    public string PhoneNumber { get; set; }

    public Contact(string name, string phoneNumber)
    {
        Name = name;
        PhoneNumber = phoneNumber;
    }

    public override string ToString()
    {
        return $"{Name}: {PhoneNumber}";
    }
}

```

Рисунок 15 – Класс контактов

```

namespace ContactSystem
{
    Ссылка 2
    class AddContactCommand : ICommand
    {
        private readonly ContactManager _manager;
        private readonly Contact _contact;

        Ссылка 1
        public AddContactCommand(ContactManager manager, Contact contact)
        {
            _manager = manager;
            _contact = contact;
        }

        Ссылка 2
        public void Execute() => _manager.AddContact(_contact);

        Ссылка 2
        public void Undo() => _manager.RemoveContact(_contact.Name);
    }
}

```

Рисунок 16 – Класс добавления контактов

```

Ссылка 2
class EditContactCommand : ICommand
{
    private readonly ContactManager _manager;
    private readonly string _oldName;
    private readonly Contact _newContact;
    private Contact _oldContact;

    Ссылка 1
    public EditContactCommand(ContactManager manager, string oldName, Contact newContact)
    {
        _manager = manager;
        _oldName = oldName;
        _newContact = newContact;
    }

    Ссылка 2
    public void Execute()
    {
        _oldContact = _manager.GetContact(_oldName);
        _manager.EditContact(_oldName, _newContact);
    }

    Ссылка 2
    public void Undo() => _manager.EditContact(_newContact.Name, _oldContact);
}

```

Рисунок 17 – Класс изменения контактов

```

Ссылка 2
class DeleteContactCommand : ICommand
{
    private readonly ContactManager _manager;
    private readonly string _contactName;
    private Contact _deletedContact;

    Ссылка 1
    public DeleteContactCommand(ContactManager manager, string contactName)
    {
        _manager = manager;
        _contactName = contactName;
    }

    Ссылка 2
    public void Execute()
    {
        _deletedContact = _manager.GetContact(_contactName);
        _manager.RemoveContact(_contactName);
    }

    Ссылка 2
    public void Undo() => _manager.AddContact(_deletedContact);
}

```

Рисунок 18 – Класс удаления контактов

```

class ContactManager
{
    private readonly Dictionary<string, Contact> _contacts = new Dictionary<string, Contact>();

    public void AddContact(Contact contact)
    {
        _contacts[contact.Name] = contact;
        Console.WriteLine($"Добавлен контакт: {contact}");
    }

    public void EditContact(string oldName, Contact newContact)
    {
        if (_contacts.ContainsKey(oldName))
        {
            _contacts.Remove(oldName);
            _contacts[newContact.Name] = newContact;
            Console.WriteLine($"Контакт '{oldName}' изменен на: {newContact}");
        }
        else
        {
            Console.WriteLine($"Контакт '{oldName}' не найден.");
        }
    }

    public void RemoveContact(string name)
    {
        if (_contacts.ContainsKey(name))
        {
            var contact = _contacts[name];
            _contacts.Remove(name);
            Console.WriteLine($"Удален контакт: {contact}");
        }
        else
        {
            Console.WriteLine($"Контакт '{name}' не найден.");
        }
    }

    Ссылка 2
    public Contact GetContact(string name)
    {
        return _contacts.ContainsKey(name) ? _contacts[name] : null;
    }

    public void ListContacts()
    {
        Console.WriteLine("Список контактов:");
        foreach (var contact in _contacts.Values)
        {
            Console.WriteLine(contact);
        }
    }
}

```

Рисунок 19 – Класс управления