Лабораторная работа №6: «Применение ООП в обработке файлов»

**Цель работы**:

Разработать классы для работы с файлами, используя принципы объектно-ориентированного программирования (инкапсуляция, наследование, полиморфизм). Реализовать чтение данных из файла, их обработку и запись результатов в другой файл.

**Задание**:

1. Создать базовый класс FileHandler с методами для чтения и записи файлов.
2. Реализовать специализированные классы-наследники:
   * TextFileHandler — для работы с текстовыми файлами.
   * CsvFileHandler — для работы с CSV-файлами.
3. Добавить обработку исключений (например, FileNotFoundException, IOException).
4. Использовать сериализацию/десериализацию объектов для сохранения данных.
5. Продемонстрировать работу классов на примере обработки данных студентов (из предыдущих лабораторных).

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text.Json;

// Базовый класс для работы с файлами

public abstract class FileHandler

{

protected string FilePath { get; set; }

public FileHandler(string filePath)

{

FilePath = filePath;

}

// Абстрактные методы для переопределения в наследниках

public abstract void WriteData(object data);

public abstract object ReadData();

// Обработка исключений

protected void HandleException(Exception ex)

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

}

// Класс для работы с текстовыми файлами

public class TextFileHandler : FileHandler

{

public TextFileHandler(string filePath) : base(filePath) { }

public override void WriteData(object data)

{

try

{

File.WriteAllText(FilePath, data.ToString());

Console.WriteLine("Данные записаны в текстовый файл.");

}

catch (Exception ex)

{

HandleException(ex);

}

}

public override object ReadData()

{

try

{

return File.ReadAllText(FilePath);

}

catch (Exception ex)

{

HandleException(ex);

return null;

}

}

}

// Класс для работы с CSV-файлами

public class CsvFileHandler : FileHandler

{

public CsvFileHandler(string filePath) : base(filePath) { }

public override void WriteData(object data)

{

if (data is List<STUDENT> students)

{

try

{

var csvLines = students.Select(s => $"{s.FullName};{s.GroupNumber};{string.Join(",", s.Grades)}");

File.WriteAllLines(FilePath, csvLines);

Console.WriteLine("Данные записаны в CSV-файл.");

}

catch (Exception ex)

{

HandleException(ex);

}

}

}

public override object ReadData()

{

try

{

var lines = File.ReadAllLines(FilePath);

var students = new List<STUDENT>();

foreach (var line in lines)

{

var parts = line.Split(';');

students.Add(new STUDENT(

parts[0],

int.Parse(parts[1]),

parts[2].Split(',').Select(int.Parse).ToArray()

));

}

return students;

}

catch (Exception ex)

{

HandleException(ex);

return null;

}

}

}

// Сериализация/десериализация объектов

public class JsonFileHandler : FileHandler

{

public JsonFileHandler(string filePath) : base(filePath) { }

public override void WriteData(object data)

{

try

{

var json = JsonSerializer.Serialize(data);

File.WriteAllText(FilePath, json);

Console.WriteLine("Данные сериализованы в JSON.");

}

catch (Exception ex)

{

HandleException(ex);

}

}

public override object ReadData()

{

try

{

var json = File.ReadAllText(FilePath);

return JsonSerializer.Deserialize<List<STUDENT>>(json);

}

catch (Exception ex)

{

HandleException(ex);

return null;

}

}

}

// Пример использования

class Program

{

static void Main()

{

// Создание списка студентов (из предыдущей работы)

var students = new List<STUDENT>

{

new STUDENT("Иванов А.А.", 101, new int[] { 5, 4, 5, 5, 4 }),

new STUDENT("Петров Б.Б.", 102, new int[] { 3, 4, 4, 3, 5 })

};

// Работа с CSV

var csvHandler = new CsvFileHandler("students.csv");

csvHandler.WriteData(students);

var csvData = csvHandler.ReadData() as List<STUDENT>;

// Работа с JSON

var jsonHandler = new JsonFileHandler("students.json");

jsonHandler.WriteData(students);

var jsonData = jsonHandler.ReadData() as List<STUDENT>;

// Вывод результатов

Console.WriteLine("\nДанные из CSV:");

csvData?.ForEach(s => Console.WriteLine($"{s.FullName} (Группа: {s.GroupNumber})"));

Console.WriteLine("\nДанные из JSON:");

jsonData?.ForEach(s => Console.WriteLine($"{s.FullName} (Группа: {s.GroupNumber})"));

}

}

#### ****Результаты выполнения****:

1. Созданы классы для работы с файлами разных форматов (текст, CSV, JSON).
2. Реализована обработка исключений при чтении/записи.
3. Продемонстрировано использование полиморфизма: методы WriteData и ReadData работают для всех типов файлов.
4. Данные студентов успешно сохраняются и загружаются из файлов.

### ****Лабораторная работа №7: «Моделирование структуры и поведения программного модуля на языке UML»****

**Объект моделирования**: Класс DecimalCounter из предоставленного кода.

**1. Диаграмма классов (Class Diagram)**



**Элементы**:

1. **Класс**DecimalCounter:
   * **Поля**:
     + \_current: int — текущее значение счетчика.
     + \_min: int — минимальное значение.
     + \_max: int — максимальное значение.
   * **Методы**:
     + Increment() — увеличивает значение.
     + Decrement() — уменьшает значение.
     + Конструкторы:
       - DecimalCounter() — по умолчанию (0-9).
       - DecimalCounter(int min, int max, int initialValue) — пользовательский.
   * **Свойство**:
     + Current: int — возвращает текущее значение.
2. **Классы исключений**:
   * CounterUnderflowException — выбрасывается при попытке уменьшить ниже \_min.
   * CounterOverflowException — выбрасывается при попытке увеличить выше \_max.
   * InvalidCounterRangeException — ошибка инициализации диапазона.

**Связи**:

* DecimalCounter **зависит** от исключений (стрелка пунктиром).

**2. Диаграмма последовательности (Sequence Diagram) для метода**Increment()

**Участники**:

* Client — вызывает метод.
* DecimalCounter — выполняет логику.
* CounterOverflowException — исключение при ошибке.

**Сценарий**:

1. Client вызывает Increment().
2. DecimalCounter проверяет условие \_current >= \_max.
3. Если условие верно:
   * Создается объект CounterOverflowException.
   * Исключение выбрасывается клиенту.
4. Если условие ложно:
   * \_current увеличивается на 1.

**3. Диаграмма состояний (State Diagram)**

**Состояния**:

* Created — объект создан.
* Valid — значение в диапазоне [\_min, \_max].
* Overflow — значение превысило \_max.
* Underflow — значение ниже \_min.

**Переходы**:

* Increment():
  + Valid → Valid (если \_current < \_max).
  + Valid → Overflow (если \_current == \_max).
* Decrement():
  + Valid → Valid (если \_current > \_min).
  + Valid → Underflow (если \_current == \_min).

**4. Диаграмма компонентов (Component Diagram)**

**Компоненты**:

* CounterModule — содержит класс DecimalCounter.
* ExceptionsModule — содержит классы исключений.

**Зависимости**:

* CounterModule зависит от ExceptionsModule.

@startuml

class DecimalCounter {

- \_current: int

- \_min: int

- \_max: int

+ Current: int

+ Increment()

+ Decrement()

+ DecimalCounter()

+ DecimalCounter(int, int, int)

}

class CounterOverflowException {

+ CounterOverflowException(string)

}

class CounterUnderflowException {

+ CounterUnderflowException(string)

}

class InvalidCounterRangeException {

+ InvalidCounterRangeException(string)

}

DecimalCounter ..> CounterOverflowException

DecimalCounter ..> CounterUnderflowException

DecimalCounter ..> InvalidCounterRangeException

@enduml

Лабораторная работа №8: «Рефакторинг кода»

**Цель**:

Улучшить читаемость, поддерживаемость и структуру кода на примере класса DecimalCounter и классов геометрических фигур (из предыдущих лабораторных работ).  
Рефакторинг включает:

1. Устранение дублирования кода.
2. Соблюдение принципов SOLID.
3. Повышение гибкости архитектуры.

**Пример 1: Рефакторинг класса**DecimalCounter

**Исходный код**:

public class DecimalCounter

{

private int \_current;

private readonly int \_min;

private readonly int \_max;

public DecimalCounter() { /\* ... \*/ }

public DecimalCounter(int min, int max, int initialValue) { /\* ... \*/ }

public void Increment()

{

if (\_current >= \_max)

throw new CounterOverflowException($"Достигнут максимум {\_max}");

\_current++;

}

public void Decrement()

{

if (\_current <= \_min)

throw new CounterUnderflowException($"Достигнут минимум {\_min}");

\_current--;

}

}

**Проблемы**:

* Дублирование проверок в Increment() и Decrement().
* Отсутствие абстракций (например, интерфейса для счетчика).

**Рефакторинг**:

// Абстракция счетчика

public interface ICounter

{

int Current { get; }

void Increment();

void Decrement();

}

// Базовый класс с общей логикой

public abstract class BaseCounter : ICounter

{

protected int \_current;

protected readonly int \_min;

protected readonly int \_max;

public BaseCounter(int min, int max, int initialValue)

{

if (min >= max)

throw new InvalidCounterRangeException("Минимум должен быть меньше максимума");

if (initialValue < min || initialValue > max)

throw new InvalidCounterRangeException("Начальное значение вне диапазона");

\_min = min;

\_max = max;

\_current = initialValue;

}

public int Current => \_current;

protected void ValidateValue(int value, int limit, Func<Exception> exceptionFactory)

{

if (value == limit)

throw exceptionFactory();

}

}

// Конкретная реализация

public class DecimalCounter : BaseCounter

{

public DecimalCounter() : base(0, 9, 0) { }

public DecimalCounter(int min, int max, int initialValue) : base(min, max, initialValue) { }

public void Increment()

{

ValidateValue(\_current, \_max, () => new CounterOverflowException($"Достигнут максимум {\_max}"));

\_current++;

}

public void Decrement()

{

ValidateValue(\_current, \_min, () => new CounterUnderflowException($"Достигнут минимум {\_min}"));

\_current--;

}

}

**Улучшения**:

* Выделена абстракция ICounter.
* Общая логика проверок вынесена в BaseCounter.
* Устранено дублирование кода.

**Результаты рефакторинга**:

1. **Снижение дублирования кода**:
   * Общая логика проверок и ввода данных вынесена в базовые классы и утилиты.
2. **Соблюдение SOLID**:
   * Принцип единой ответственности: валидация, трансформации, ввод данных выделены в отдельные компоненты.
   * Принцип открытости/закрытости: добавление новых фигур не требует изменения существующего кода.
3. **Улучшенная читаемость**:
   * Имена методов и классов стали более информативными (например, ITransformable вместо Line с методом Rotate).