Лабораторная работа №1: Создание и использование классов.

Цель работы:

- Знакомство с базовыми принципами ООП.
- Знакомство с основами создания классов.

Теоретическая информация.

Объектно-ориентированное программирование (ООП) — это парадигма программирования, в которой программа строится вокруг объектов.

ООП возникло как ответ на необходимость улучшения структуры и организации программ, а также на стремление сделать программирование более интуитивно понятным и управляемым. Процедурное программирование, основанное на разделении программы на функции и процедуры, становилось неэффективным для больших и сложных систем, так как это часто приводило к дублированию кода и запутанности.

Концепция ООП же основана на моделировании программы в терминах реальных сущностей или объектов, что сделало код более интуитивно понятным, легче расширяемым и поддерживаемым.

Класс — это основной инструмент в технологии ООП. Класс представляет собой шаблон или модель для создания объектов. Класс определяет общие характеристики (атрибуты) и поведение (методы) объектов, которые на его основе могут быть созданы. Класс является логической абстракцией. Физическое представление класса появится в программе лишь после того, как будет создан **объект** (экземпляр) этого класса.

Основная идея при выделении классов заключается в обобщении некоторых сущностей, которые обладают схожими параметрами и выполняют одинаковые функции. К примеру, мы делаем программу для учета книг в библиотеке. Нашим основным **классом** в данном случае будет как раз «Книга», которая будет обладать некоторыми атрибутами – названием, автором, количеством страниц. А уже какие-то записи о существующих в библиотеке книгах будут экземплярами **объектов** данного класса:

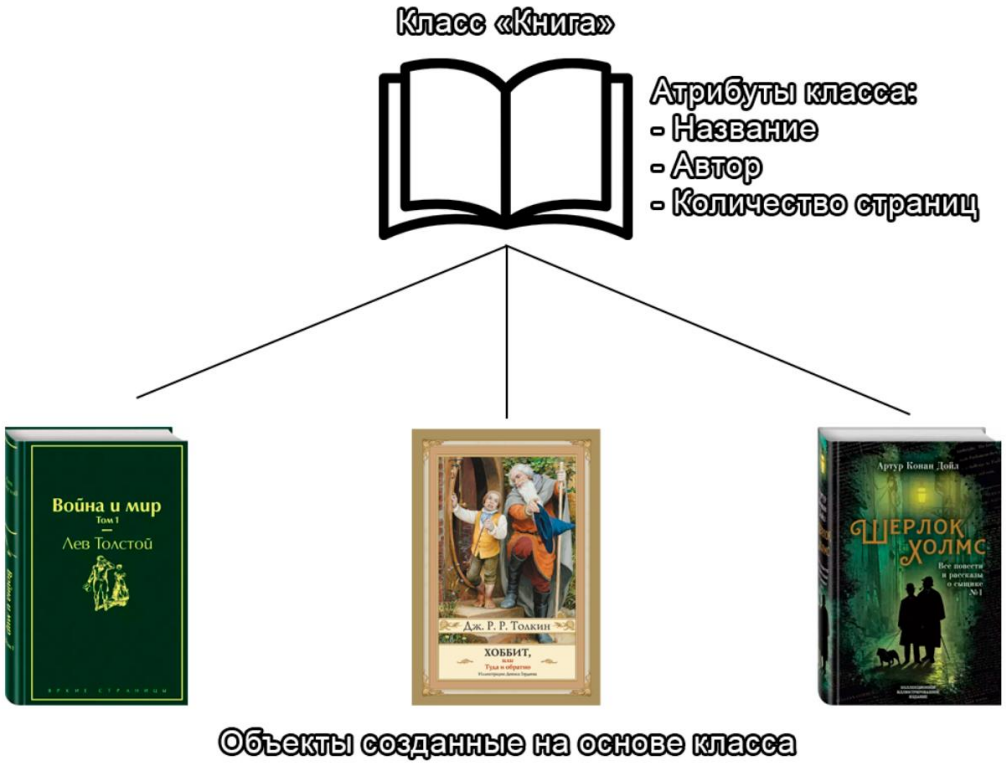

Рисунок 1 – Визуальный пример класса «Книга» и объектов, созданных на его основе

Данная реализация решает проблему модифицируемости нашей программы – при появлении в библиотеке новой книги требуется всего лишь создать новый экземпляр объекта с нужными данными.

На языке программирования C# класс создается с помощью ключевого слова **class**. Ниже приведена общая структура класса, содержащая несколько атрибутов и методов:

```
//тело класса
public class Person
    {
        //атрибуты класса
        int age;
        string first_name;
        string second_name;
        double height;

        //методы класса
        public string getFullName()
        {
            //метод возвращает строку из комбинации имени и фамилии
            return $"{first_name + second_name}";

        }

        public double getHeightInIches()
        {
            //метод возвращает рост переведенный из сантиметров в дюймы
            return height * 2.54;
        }
    }
```

В данном примере у класса под названием *Person* существует четыре различных атрибута разных типов данных и два метода которые используют эти атрибуты.

Перед каждым объявлением поля или метода указывается **модификатор доступа**. Модификатор доступа определяет тип разрешенного доступа. Существует несколько различных модификаторов, но в данной работе мы будем использовать следующие два модификатора:

– `public` - данный модификатор доступа делает атрибут или метод доступным всем. То есть к публичным параметрам можно обращаться из любой части кода, в том числе из других классов.
– `private` - данный модификатор доступа позволяет обращаться к атрибутам и методам класса только изнутри самого класса. Если не указывается модификатор доступа, то стандартным выбирается модификатор private.

В данном примере класса его атрибуты обозначены как *приватные*. Но методы, которые реализует данный класс являются *публичными* и через эти публичные методы мы имеем возможность взаимодействовать с *приватными* полями.

Добавление класса осуществляется в контекстном меню проекта: необходимо выбрать пункт «Добавить – Класс...» (или воспользоваться комбинацией клавиш Shift+Alt+C). В появившемся диалоговом окне необходимо ввести имя файла – данное имя так же служит и названием для реализуемого класса.

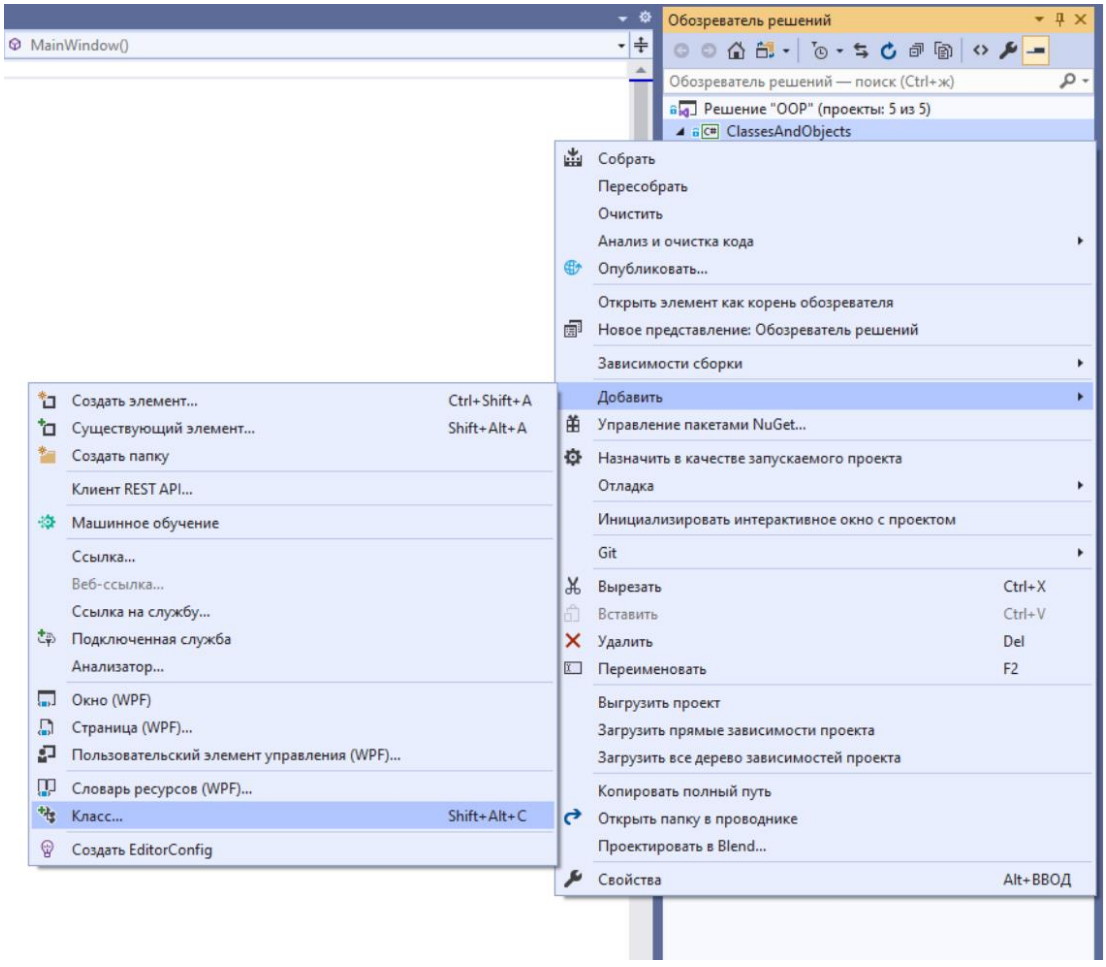

Рисунок 2 – Создание нового класса в проекте

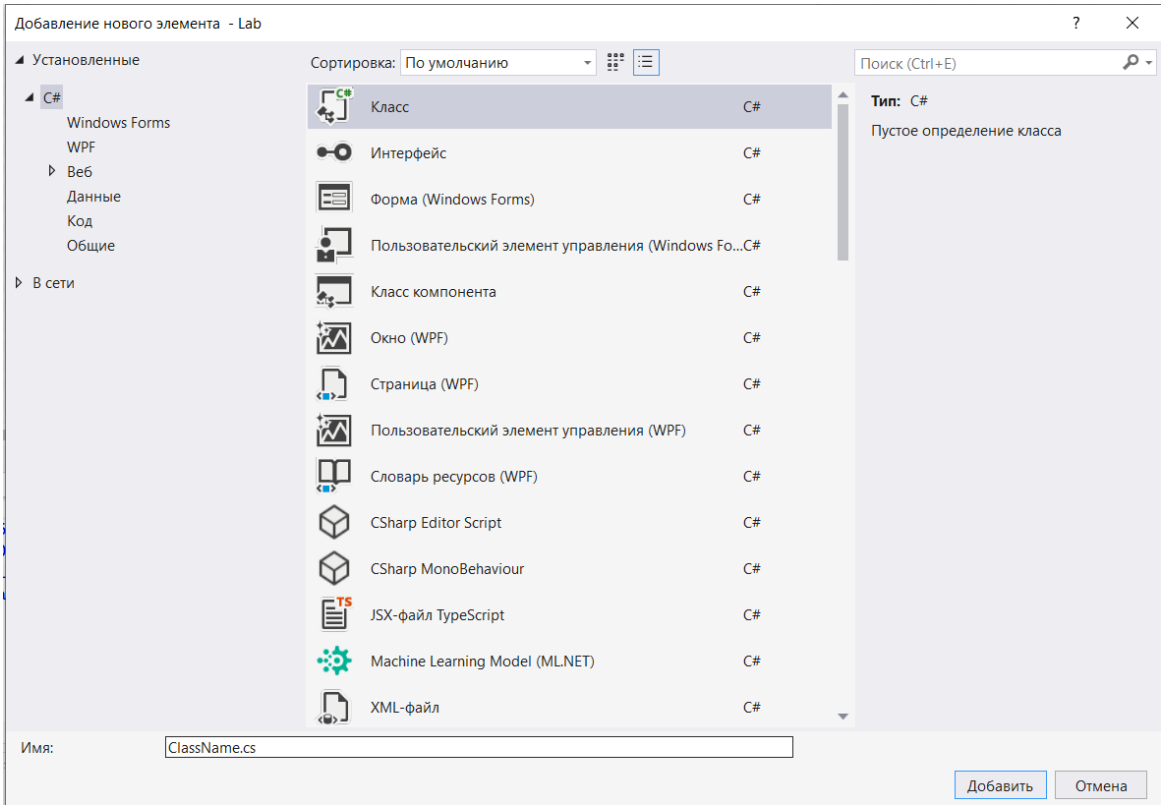

Рисунок 3 – Диалоговое окно добавления нового элемента

Одним из самых важных элементов класса является его конструктор. **Конструктор** – метод, инициализирующий объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу, за исключением явно указываемого возвращаемого типа. Конструктор класса вызывается автоматически при использовании оператора **new**.

```
//тело класса
public class Person
    {
        //атрибуты класса
        int age;
        string first_name;
        string second_name;
        double height;

        //конструктор класса
        public Person(int Age, string FName, string SName, double Height)
        {
            age = Age; //присваивание атрибутам передаваемые значения
            first_name = FName;
            second_name = SName;
            height = Height;
        }
    }
```

Теперь в основной программе мы можем создать новый экземпляр класса используя данный конструктор:

```
public MainWindow()
        {
            InitializeComponent();

            //создание экземпляра класса
            Person person1 = new Person(18, "Иван", "Иванов", 179.56);
        }
```

Задание №1. Рисование геометрических фигур.

Первым заданием лабораторной работы является разработка программы для рисования фигур.

Функционал программы:

– Рисование двух видов фигур на холсте – треугольника и прямоугольника. У пользователя должна быть возможность либо задать размеры фигур, либо они генерируются со случайным размером.
– Точка, треугольник и прямоугольник должны быть реализованы в качестве классов.
– В программе должен присутствовать функционал для перемещения фигур по холсту по обоим осям координат.

Задачи:

- Реализовать программу для рисования геометрических фигур используя представленные примеры.
- Создать свой класс для четырехугольника. Создавать четырехугольник лучше всего через начальную точку и расстояния его длинны и ширины.
- Реализовать функционал для создания треугольника и квадрата со случайными точками или заданными пользователем
- Реализовать функционал для перемещения фигур по сцене.

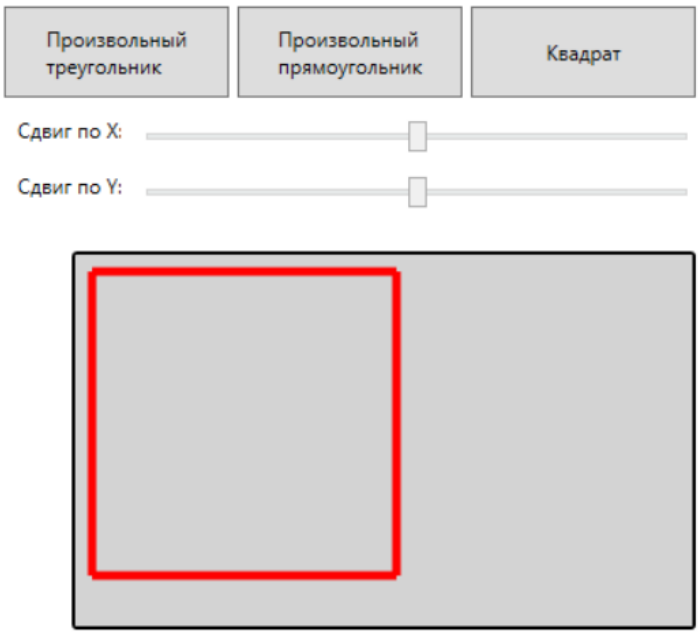

Рисунок 4 – Пример интерфейса программы

Для рисования геометрических фигур в окне необходимо создать «холст» в области окна. В качестве «холста» можно использовать элемент управления **Canvas**. В данном примере холст имеет название *“Scene”*.

```
<Grid>
      <Canvas   x:Name="Scene"   HorizontalAlignment="Left"   Height="400"     Margin="15,0,0,0"
VerticalAlignment="Center" Width="680"/>
</Grid>
```

Для того чтобы рисовать геометрические фигуры предлагается использовать отрисовку по линиям. Для этого можно использовать класс **Line.** Рисование линии с помощью класса Line состоит из следующих шагов:

- создание объекта (экземпляра) класса Line;
- задание визуальных параметров линии: цвета и толщины;
- задание относительных координат начала и конца линии;
- добавление линии в холст.

Для того чтобы хранить информацию о координатах создадим класс точки *Point2D*. Данный класс логически будет представлять собой точку с координатами X и Y:

```
public class Point2D
    {
        //Атрибуты класса
        private int X;
        private int Y;

        //Конструктор класса
        public Point2D(int x, int y)
        {
            //this используется для однозначного указания на атрибуты класса так как
переменные имеют одинаковые имена
            this.X = x;
            this.Y = y;
        }


        //Методы для получения координат
        public int getX()
        {
            return X;
        }

        public int getY()
        {
            return Y;
        }

        //Методы для изменения координат
        public void addX(int x)
        {
            X += x;
        }

        public void addY(int y)
        {
            Y += y;
        }
    }
```

Теперь, когда у нас есть готовый класс, который содержит в себе информацию о координатах его можно использовать в другом классе, который будет реализовывать структуру фигуры. Свойство, когда один класс использует другой класс в качестве атрибута называется **агрегацией**.

Создадим класс треугольника *Triangle*:

```
public class Triangle
    {
        //Атрибуты класса
        private Point2D p1;
        private Point2D p2;
        private Point2D p3;

        //Конструктор класса
        public Triangle(Point2D p1, Point2D p2, Point2D p3)
        {
            this.p1 = p1;
            this.p2 = p2;
            this.p3 = p3;
        }

        public Point2D getP1()
        {
            return p1;
        }

        public Point2D getP2()
        {
            return p2;
        }

        public Point2D getP3()
        {
            return p3;
        }
    }
```

Для того чтобы изменять координаты воспользуемся методом изменения координат который был реализован в классе точки и применим его ко всем точкам внутри класса фигуры:

```
public void addX(int X)
        {
            p1.addX(X);
            p2.addX(X);
            p3.addX(X);
        }

public void addY(int Y)
        {
            p1.addY(Y);
            p2.addY(Y);
            p3.addY(Y);
        }
```

Даже учитывая одинаковые названия методов программа однозначно вызывает метод для конкретного класса – в данном случае класс Point2D вызывает свой метод addX/addY так как он вызывается от экземпляра класса Point2D.

Полученные классы можно представить в виде следующих UML-диаграмм:

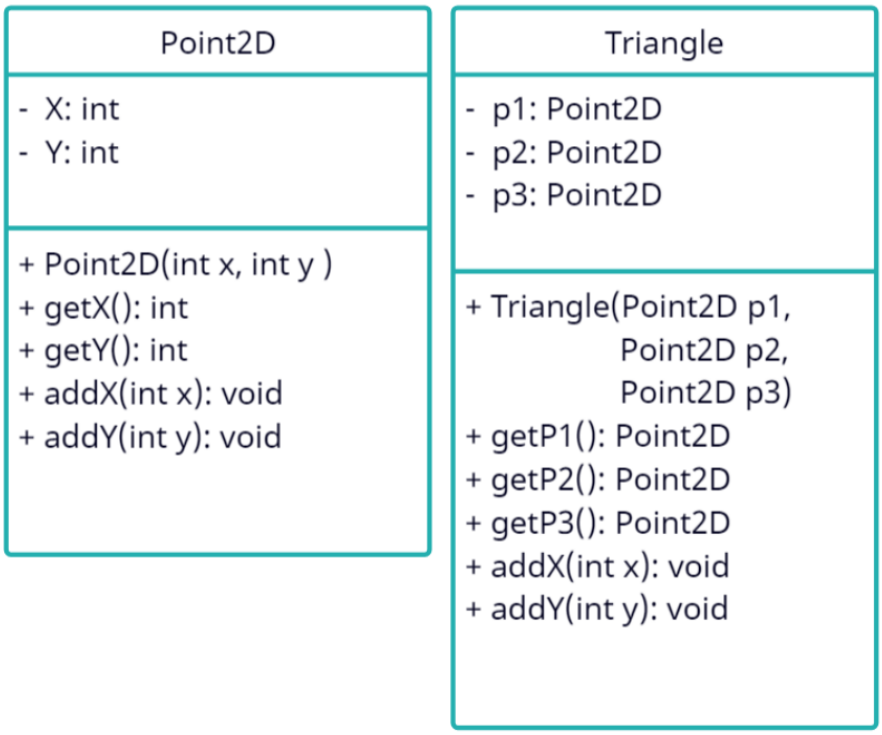

Рисунок 5 – UML-диаграммы классов Point2D и Triangle

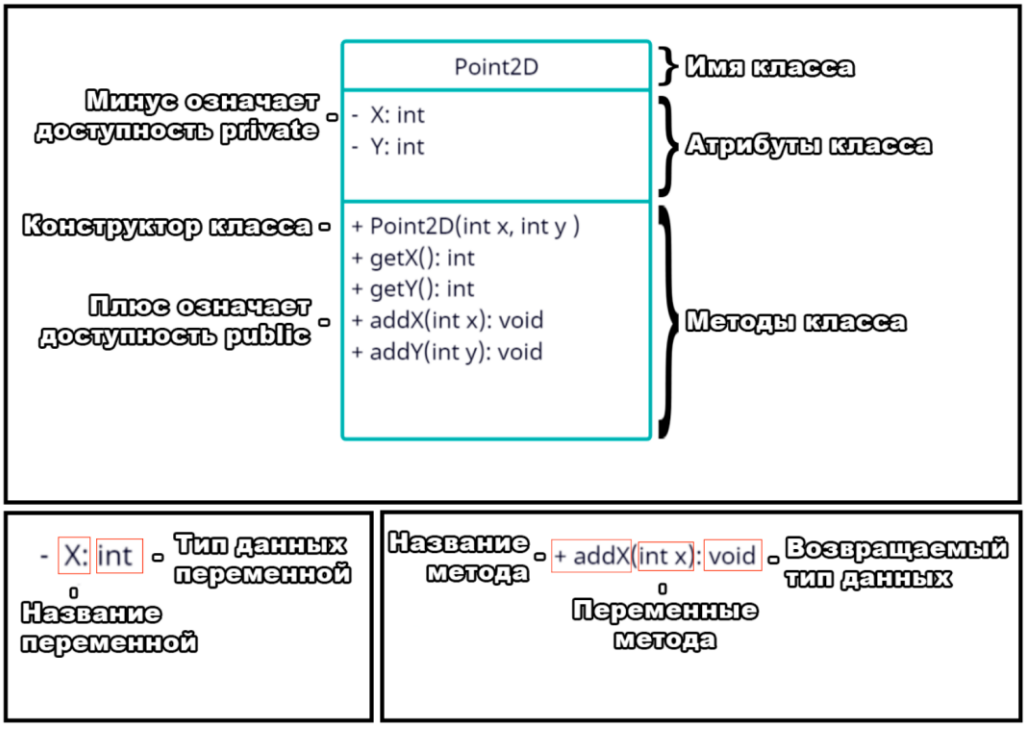

Рисунок 6 – Как читать UML-диаграммы

Далее реализуем функцию в основном коде программы для рисования линии по двум точкам, в данном примере используется встроенный класс для рисования линии:

```
//функция в основном теле программы
public void DrawLine(Point2D p1, Point2D p2)
    {
        //Создание новой линии
        Line line = new Line();
        //Цвет и толщина линии
        line.Stroke = Brushes.Red;
        line.StrokeThickness = 3;

        //Установка координат линии из координат точек Point2D
        line.X1 = p1.getX();
        line.Y1 = p1.getY();
        line.X2 = p2.getX();
        line.Y2 = p2.getY();

        //Добавление линии в Canvas
        Scene.Children.Add(line);
    }
```

Создадим новый треугольник со случайными координатами:

```
Triangle tr;
Random rnd = new Random();

public MainWindow()
    {
        //Создание треугольника со случайными координатами
        Point2D p1 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0,
(int)Scene.Height));
        Point2D p2 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0,
(int)Scene.Height));
        Point2D p3 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0,
(int)Scene.Height));
        tr = new Triangle(p1, p2, p3);
    }
```

Функция рисования треугольника будет выглядеть следующим образом:

```
public void DrawTriangle(Triangle tr)
    {
        //Отрисовка треугольника с помощью функции отрисовки линии
        DrawLine(tr.getP1(), tr.getP2());
        DrawLine(tr.getP2(), tr.getP3());
        DrawLine(tr.getP3(), tr.getP1());
    }
```

Для того чтобы очистить сцену от линий можно использовать следующую функцию:

```
public void ClearScene()
    {
         //Очистка Canvas от всех объектов
        Scene.Children.Clear();
    }
```

Задание №2. Редактор противников.

В течении нескольких лабораторных работ будет производится разработка игры-кликера. Первым этапом разработки является редактор противников.

Функционал программы:

- Загрузка всех изображений из указанной папки в качестве иконок.
- Форматирование иконок под единый размер, а также их визуализация в программе.
- Добавление/удаление противников в список.
- Загрузка/сохранение списка противников в формате JSON.

Задачи:

- Реализовать программу согласно предложенному функционалу.
- Разработать систему классов для программы.
- Класс иконки должен реализовывать

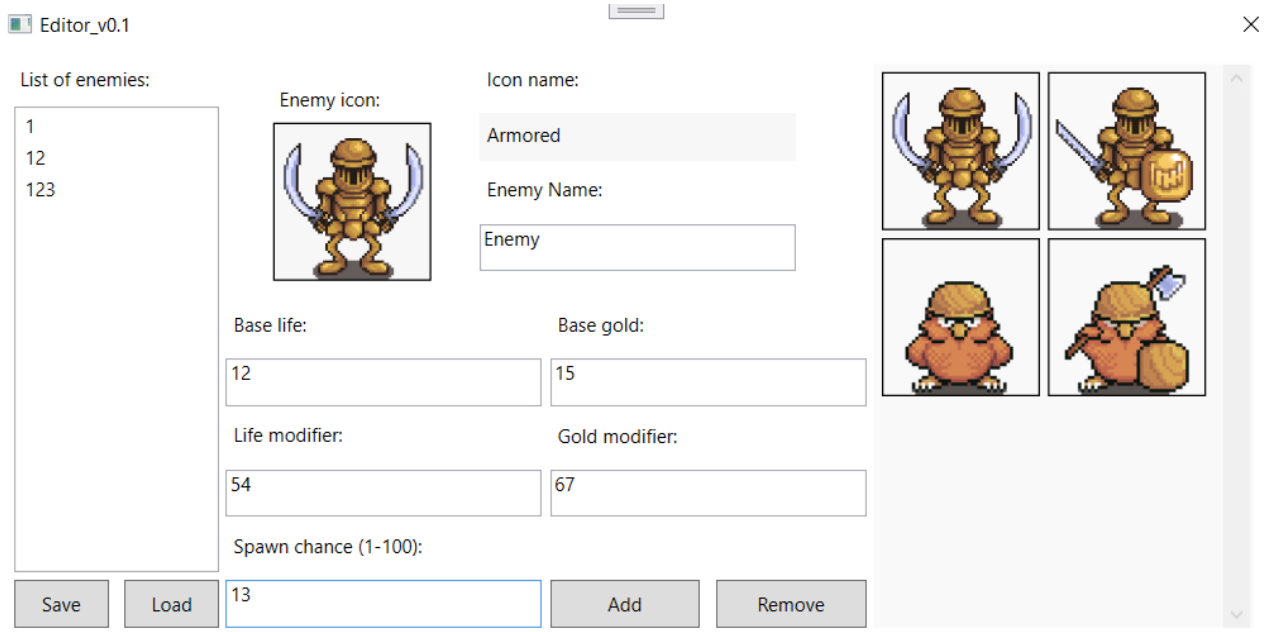

Рисунок 7 – Пример интерфейса программы

В качестве класса противника можно использовать следующую структуру, совпадающую со структурой из примера:

```
public class CEnemyTemplate
    {
        //Название противника
        string name;
        //Название иконки
        string iconName;

        //Атрибуты здоровья
        int baseLife;
        double lifeModifier;

        //Атрибуты золота за победу над противником
        int baseGold;
        double goldModifier;

        //Шанс на появление
        double spawnChance;
    }
```

Так как в данном примере атрибуты реализованы как приватные требуется реализация функций для получения данных значений. Получать значения эти атрибуты должны через конструктора, который так же требуется реализовать в классе:

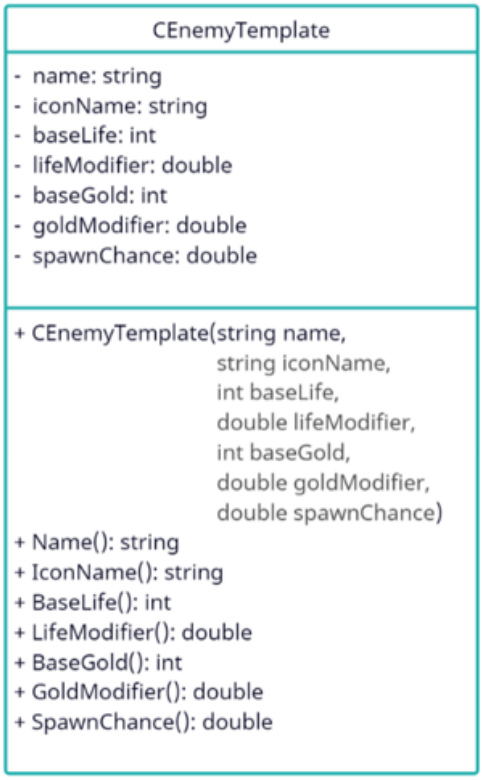

Рисунок 8 – UML-диаграмма класса противника

Класс противника требуется только для хранения информации о противнике и её получении. Для реализации функционала работы с противником следует реализовать отдельный класс, который будет отвечать за хранение списка противников, добавления и удаления, а также для сохранения и загрузки:

```
public class CEnemyTemplateList
    {
        //Список противников из класса CEnemyTemplate
        List<CEnemyTemplate> enemies;

        public CEnemyTemplateList()
        {
            enemies = new List<CEnemyTemplate>();
        }

    }
```

Для того чтобы сохранить экземпляр класса в формат JSON для начала требуется подключить библиотеку для работы с данным форматом. Для этого требуется перейти во вкладку «Управление пакетами NuGet», затем во вкладке «Обзор» найти пакет «System.Text.Json» и установить последнюю версию.

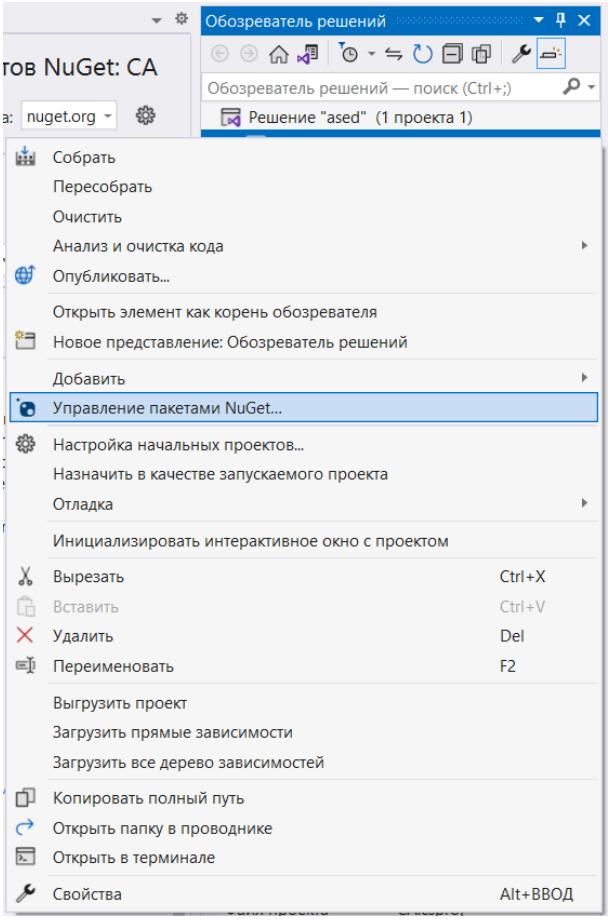

Рисунок 9 – Вкладка управления пакетами NuGet

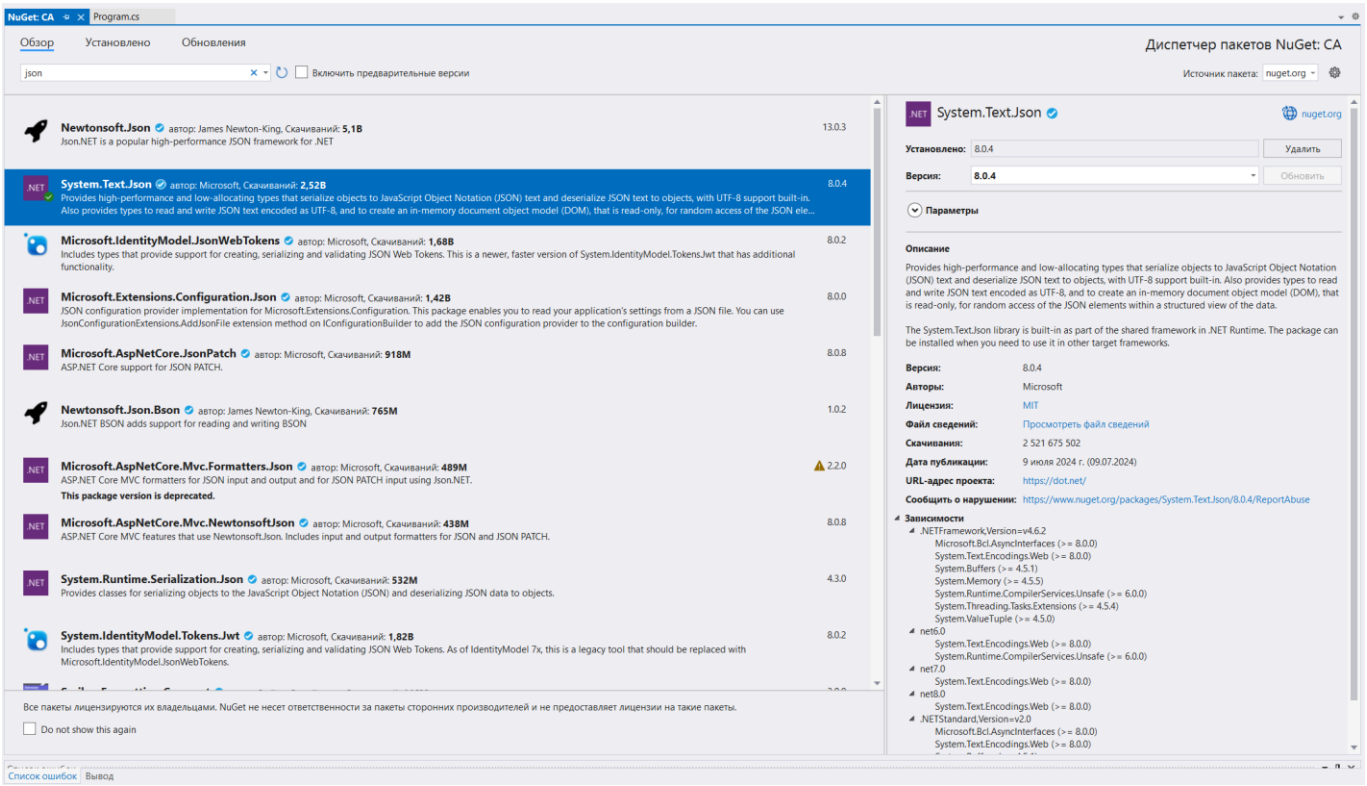

Рисунок 10 – Пакет System.Text.Json во вкладке обзора

Для того чтобы сохранить список класса в JSON требуется добавить ключ [JsonInclude] перед атрибутами, которые будут сохраняться в данный формат, затем нужно сериализовать список с помощью встроенной в пакет функции JsonSerializer.Serialize, затем сериализованный список можно сохранить в файл:

```
using System;
using System.Collections.Generic;
using System.Text.Json;
using System.IO;
using System.Text.Json.Serialization;

namespace Program1
{
    public class Person
    {
        [JsonInclude] //Атрибут будет сохранен в json
        int age;
        [JsonInclude]
        string first_name;
        [JsonInclude]
        string second_name;
        [JsonInclude]
        double height;
        public Person(int Age, string FName, string SName, double Height)
        {
            age = Age;
            first_name = FName;
            second_name = SName;
            height = Height;
        }

        public int getAge() { return age; }
        public string getFirstName() { return first_name;}
        public string getSecondName() { return second_name;}
        public double getHeight() { return height;}
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            // Создаем список экземпляров класса Person
            List<Person> people = new List<Person>();
            people.Add(new Person(20, "Ivan", "Ivanov", 177.65));
            people.Add(new Person(30, "Aleksey", "Alekseevich", 166.99));
            people.Add(new Person(25, "Oleg", "Olegovich", 180.01));

            // Сериализация списка в JSON
            string jsonString = JsonSerializer.Serialize(people);

            // Сохранение JSON в файл
            File.WriteAllText("people.json", jsonString);
        }
    }
}
```

То как выглядит структура сохраненного файла можно посмотреть, открыв его в Visual Studio:

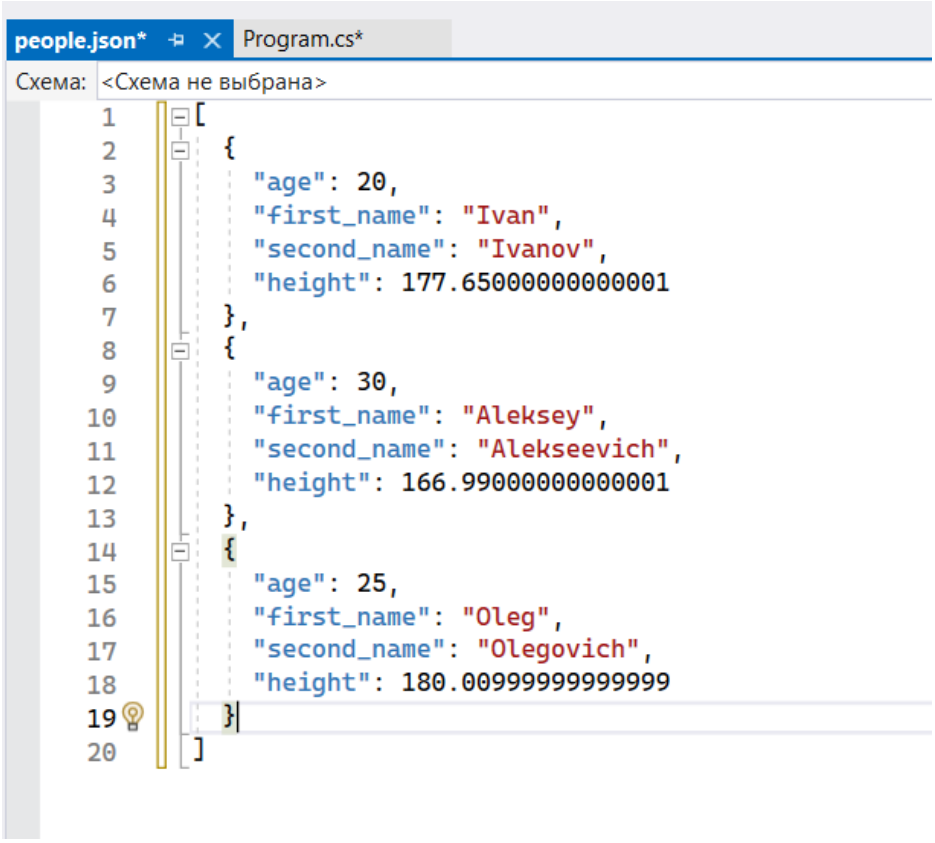

Рисунок 11 – Структура сохраненного JSON файла

Для считывания будет использоваться «ручная» конвертация в объект класса так как в данной работе используются приватные атрибуты, которые в чистом виде не могут быть конвертированы в JSON и обратно:

```
using System;
using System.Collections.Generic;
using System.Text.Json;
using System.IO;
using System.Text.Json.Serialization;

namespace Program1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            // Чтение JSON из файла
            string jsonFromFile = File.ReadAllText("people.json");
            List<Person> people = new List<Person>();

            // Парсинг JSON
            JsonDocument doc = JsonDocument.Parse(jsonFromFile);
            //Добавление новой записи в список класса из json
            foreach (JsonElement element in doc.RootElement.EnumerateArray())
            {
                int age = element.GetProperty("age").GetInt32();
                string firstName = element.GetProperty("first_name").GetString();
                string secondName = element.GetProperty("second_name").GetString();
                double height = element.GetProperty("height").GetDouble();
                // Создание нового экземпляра класса Person с помощью конструктора
                Person person = new Person(age, firstName, secondName, height);
                // Добавление объекта в список
                people.Add(person);
            }
            // Вывод данных на экран
            foreach (var person in people)
            {
                Console.WriteLine($"Age: {person.Age()}, Name: {person.FirstName()}
{person.SecondName()}, Height: {person.Height()}");
            }
        }
    }
}
```

Помимо функций сохранения и загрузки в классе следует реализовать функционал по добавлению и удалению экземпляров противников.

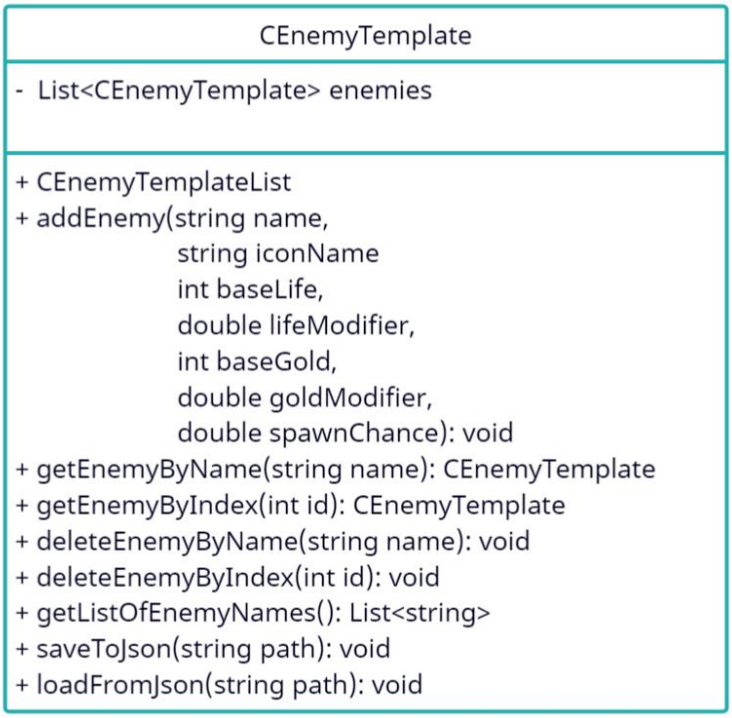

Рисунок 12 – UML-диаграмма класса для работы с шаблоном противника

Для организации работы с иконками противников будем использовать объектно-ориентированный подход, разделив данные и логику их обработки. Потребуется один класс для хранения информации:

EnemyIcon – класс для хранения данных об одной иконке. Включает в себя два свойства: Name (имя файла с расширением) и ImagePath (полный путь к файлу изображения).

```
// класс представляющий иконку врага
public class EnemyIcon
{
    // имя иконки с расширением
    public string Name { get; set; }
    // полный путь до иконки
    public string ImagePath { get; set; }
}
```

Процесс загрузки и выбора иконки состоит из нескольких шагов.

1. Выбор папки и загрузка данных об иконках.

Для того чтобы пользователь мог указать, где находятся изображения иконок, используется диалоговое окно **OpenFolderDialog**. **OpenFolderDialog** позволят выбрать **путь**, а не конкретный файл. После выбора папки вызывается метод **LoadIconsFromFolder**, который сканирует указанную директорию.

Метод **LoadIconsFromFolder** получает все файлы с расширением .png и для каждого найденного файла создает объект класса **EnemyIcon**, заполняя его свойства **Name** и **ImagePath**. Все созданные объекты добавляются в список **List<EnemyIcon> enemyIcons.**

```
public void LoadIconsFromFolder(string path)
{
    //фильтр расширения изображения
    string filter = "*.png";

    //получение массива строк содержащих пути до изображений
    string[] files = Directory.GetFiles(path, filter);

    //перебор всех полученных путей
    //в file содержится путь до изображения с расширением .png
    foreach (string file in files)
    {
        enemyIcons.Add(
            new EnemyIcon
                {
                // получение имени файла с расширением
                Name = System.IO.Path.GetFileName(file),
                // получение полного пути до файла
                ImagePath = file
                }
            );
    }
}
```

2. Отображение иконок в интерфейсе

Для отображения иконок используется элемент **ListBox**. Вместо сложного механизма привязки данных *(но, если хотите, можете обратиться к лабораторным работам 3 и 4 прошлого семестра и сделать привязку самостоятельно)*, мы добавим изображения в **ListBox** программно. Для этого необходимо перебрать список **enemyIcons** и для каждого элемента создать UI-элемент **Image**. В свойство **Source** этого элемента загружается изображение по пути, хранящемуся в **icon.ImagePath**.

Созданный Image добавляется в коллекцию **Items** элемента **ListBox**.

```
// заполнение ListBox иконками
foreach (EnemyIcon icon in enemyIcons)
{
    // создание элемента Image для отображения иконки
    Image image = new Image()
    {
        // установка источника изображения
        Source = new BitmapImage(
            new Uri(icon.ImagePath) // Uri это универсальный идентификатор ресурса
(Uniform Resource Identifier
            // который указывает на местоположение ресурса, в данном случае на путь
к файлу изображения
        ),
        Height = 64, // высота изображения
    };
    IconsListBox.Items.Add(image); // добавление изображения в ListBox
}
```

3. **Обработка выбора иконки**

   Чтобы отследить, какую иконку выбрал пользователь, необходимо подписаться на событие **SelectionChanged** у **ListBox**. В XAML-разметке это выглядит так:

```
<ListBox
    Width="400"
    Margin="10 20"
    Name="IconsListBox"
    SelectionChanged="IconsListBox_SelectionChanged"/>
```

В обработчике события **IconsListBox_SelectionChanged** мы получаем доступ к выбранному элементу. Так как мы вручную добавляли в **ListBox** объекты типа **Image**, то и **SelectedItem** будет являться объектом **Image**.

Из свойства **Source** выбранного изображения можно получить полный путь к файлу, из которого, в свою очередь, извлекается имя файла. Это имя сохраняется в объект **someEnemy** для дальнейшего использования.

```
// обработчик события изменения выбора в ListBox
private void IconsListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    // приведение sender к типу ListBox
    ListBox iconHolder = sender as ListBox;

    // проверка что выбранный элемент является изображением
    // и что элемент не равен null
    if (iconHolder.SelectedItem is Image selectedImage && iconHolder.SelectedItem != null)
    {
        // получение имени файла из источника изображения
        // так как Source это Uri, то для получения имени файла
        // нужно преобразовать его в строку и использовать Path.GetFileName.

        // ListBox хранит в себе объекты типа Image
        // selectedImage.Source.ToString() возвращает полный путь до изображения
        string iconName = System.IO.Path.GetFileName(selectedImage.Source.ToString());
        // присвоение имени иконки в шаблон врага
        someEnemy.IconName = iconName;
    }
}
```

Пример работы:

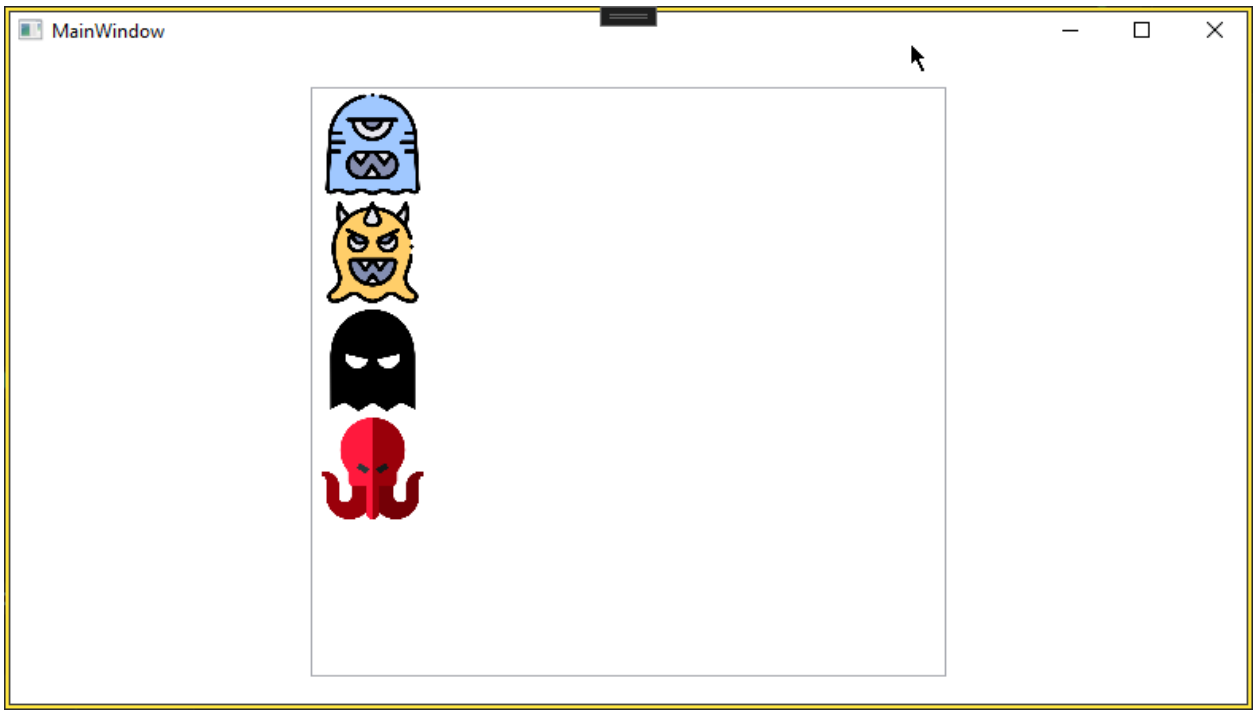

Рисунок 13.

Отображение «главной иконки» противника вы также можете реализовать через **Image. Image** может быть размещен в окне **xaml** редактора.

```
<Image Name="MainEnemyIcon"/>
```

Справочная информация

Открытие диалогового окна сохранения:

```
private void Button_Click(object sender, RoutedEventArgs e)
        {
            //создание диалога
            OpenFileDialog dlg = new OpenFileDialog();
            //настройка параметров диалога
            dlg.FileName = "Document"; // Default file name
            dlg.DefaultExt = ".txt"; // Default file extension
            dlg.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension
                                                        //вызов диалога
            dlg.ShowDialog();
            //получение выбранного имени файла
            lb1.Content = dlg.FileName;
        }
```

Открытие диалогового окна загрузки:

```
private void Button_Click(object sender, RoutedEventArgs e)
        {
            //создание диалога
            SaveFileDialog dlg = new SaveFileDialog();
            //настройка параметров диалога
            dlg.FileName = "Document"; // Default file name
            dlg.DefaultExt = ".txt"; // Default file extension
            dlg.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension
                                                        //вызов диалога
            dlg.ShowDialog();
```

```
            //получение выбранного имени файла
            lb1.Content = dlg.FileName;
        }
```