

Лабораторная работа №2: Применение классов при создании архитектуры программы.

Цель работы:

- Применение знаний о создании классов для разработки ПО.

Задачи:

- Реализовать программу согласно предложенному функционалу.
- Разработать систему классов для программы: классы для шаблона противника и иконки, классы для работы со списком шаблонов противников и иконок.
- Реализовать систему сохранения и загрузки списка противников.

Функционал программы:

- При старте программы должна происходить загрузка всех изображений из указанной в коде программы папки в качестве иконок.
- Добавление/удаление противников в список.
- Загрузка/сохранение списка противников в формате JSON.

Задание. Редактор противников.

В качестве задания на лабораторную работу требуется написать программу для редактирования списка противников для игры-кликера, которая будет реализовываться в лабораторной работе №3.

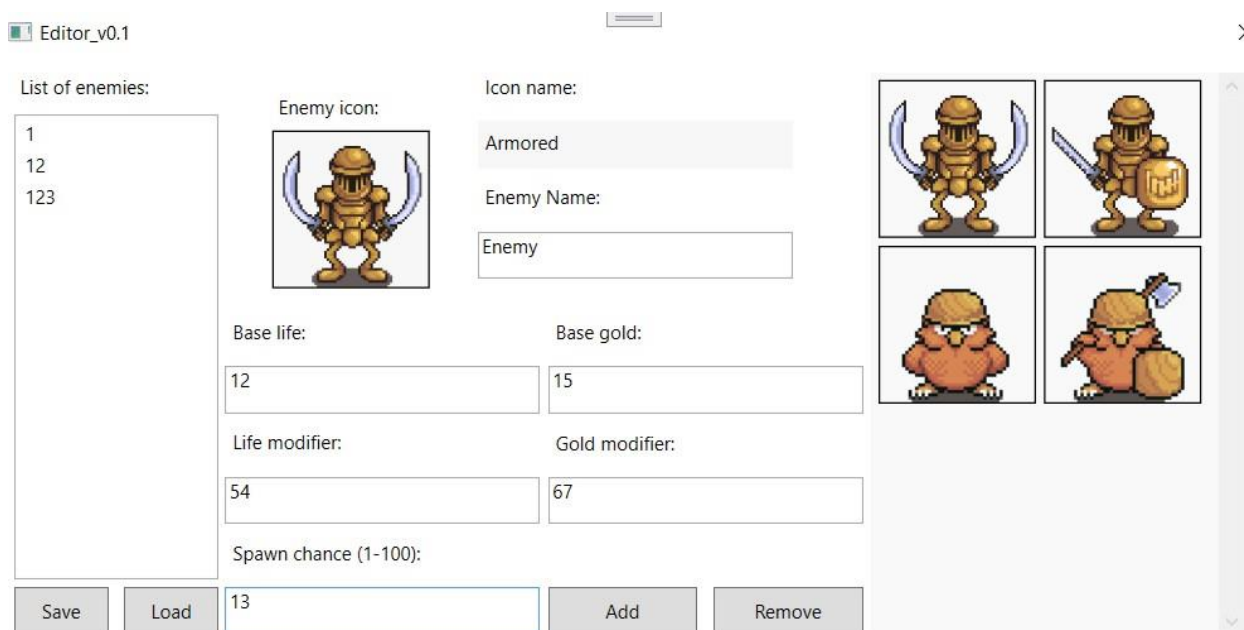


Рисунок 1 – Пример интерфейса программы

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 конструктор, который так же требуется реализовать в классе:

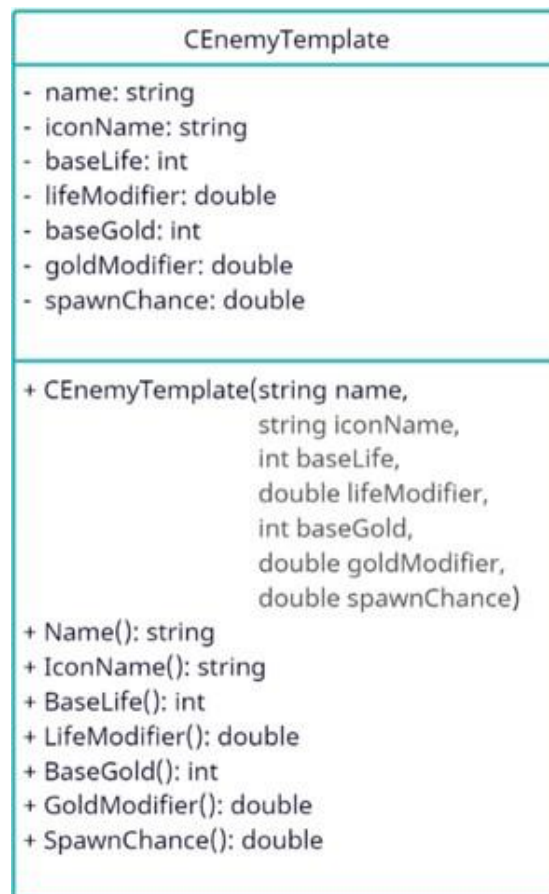


Рисунок 2 – UML-диаграмма класса противника

Обратите внимание что требуется так же реализовать функции получения данных из класса, а также конструктор.

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:

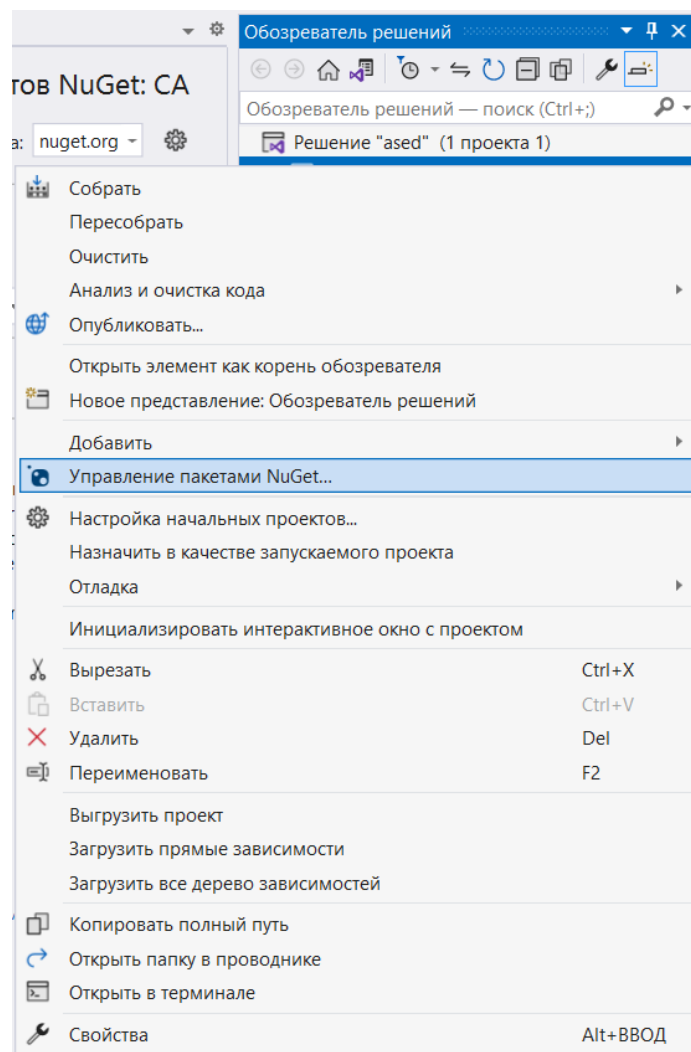


Рисунок 3 – Вкладка управления пакетами NuGet

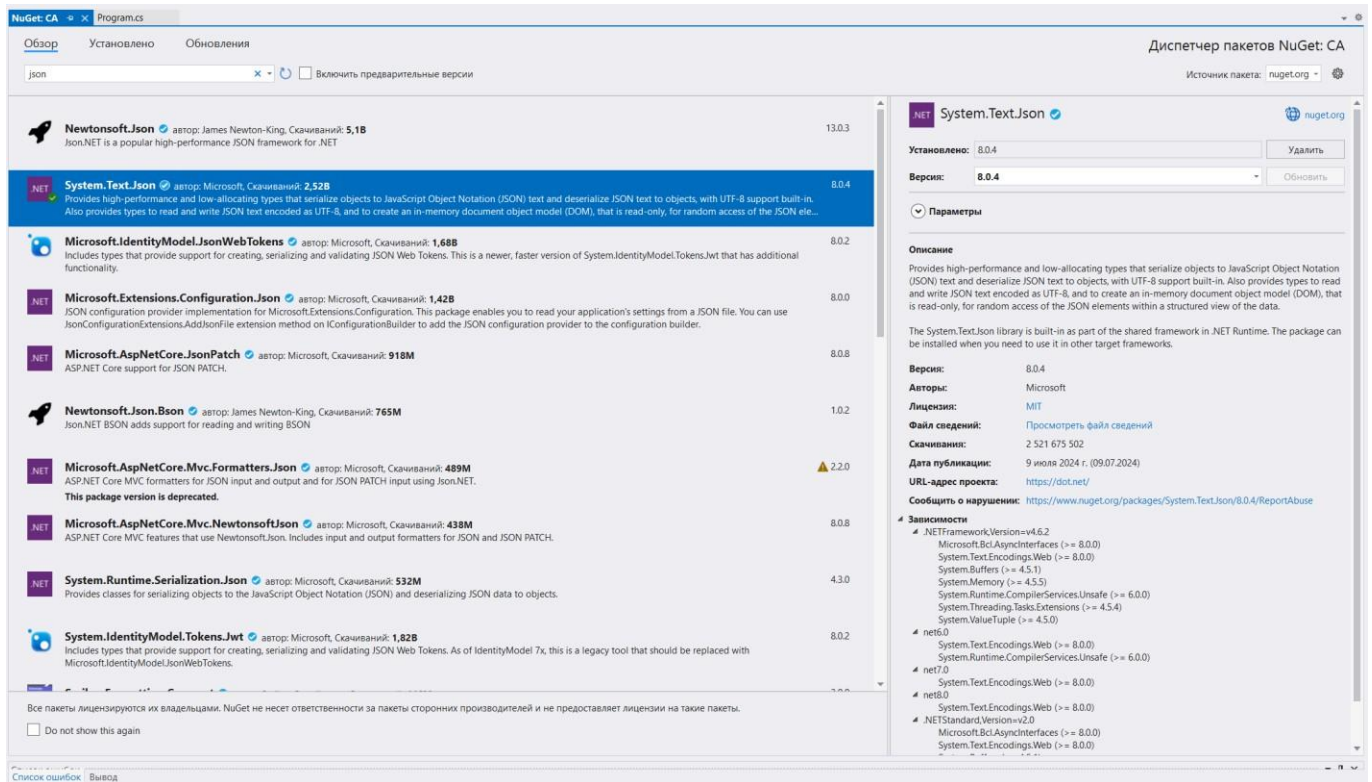


Рисунок 4 – Пакет System.Text.Json во вкладке обзора

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;}
    }

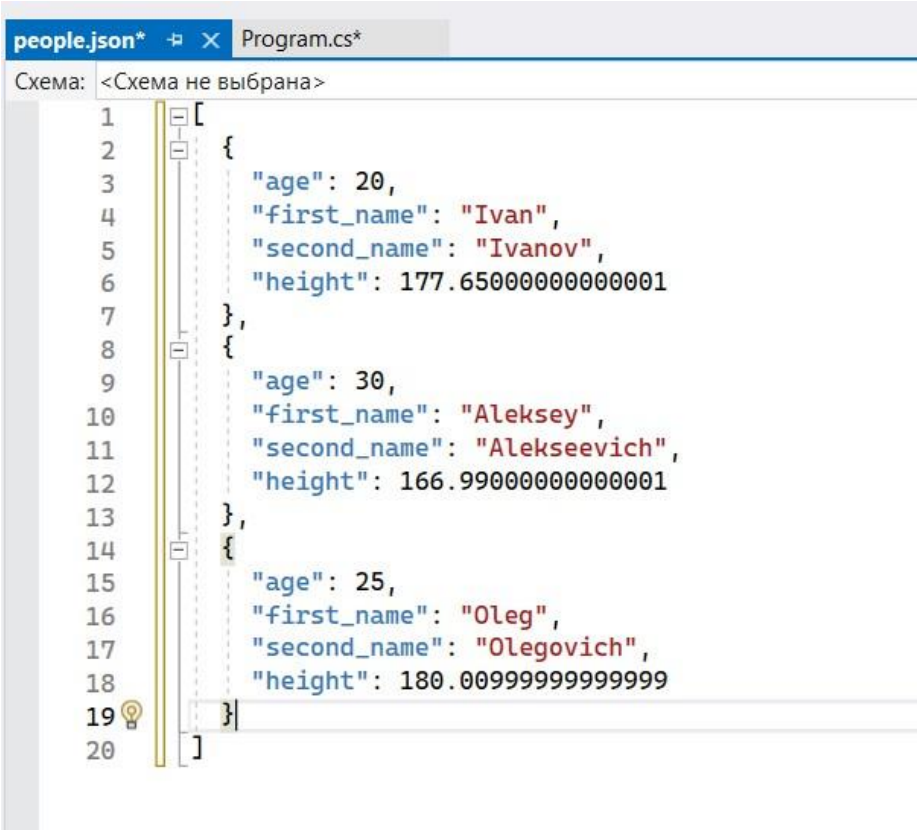
    internal class Program
    {
        static void Main(string[] args)
        {
            // Создаем список экземпляров класса Person
            List<Person> people = new List<Person>();
            people.Add(new Person(20, "Ivan", "Ivanov", 177.65));
            people.Add(new Person(30, "Aleksey",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

Важно: на представленном коде показан пример сохранения класса Person в Json в консольном приложении. Для вашей работы требуется понять принцип работы с Json и адаптировать его под реализуемые в программе классы.

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



```
1  [
2  {
3      "age": 20,
4      "first_name": "Ivan",
5      "second_name": "Ivanov",
6      "height": 177.65000000000001
7  },
8  {
9      "age": 30,
10     "first_name": "Aleksey",
11     "second_name": "Alekseevich",
12     "height": 166.99000000000001
13  },
14  {
15     "age": 25,
16     "first_name": "Oleg",
17     "second_name": "Olegovich",
18     "height": 180.00999999999999
19  }
20 ]
```

Рисунок 5 – 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. Тут так же показан пример в консольном приложении с классом Person:

```
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.

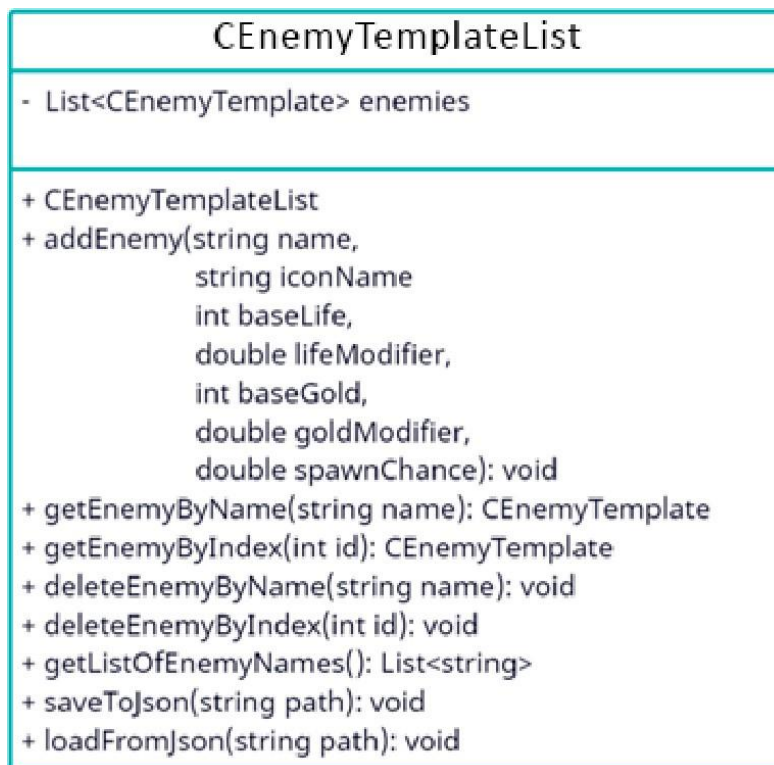


Рисунок 6 – UML-диаграмма класса для работы с шаблоном противника

Для хранения и использования иконок рекомендуется так же использовать два класса:

- *Icon* – для хранения данных о спрайте. В качестве атрибутов можно использовать размеры иконки, имя, позицию и само изображение, представленное в виде Rectangle.
- *IconList* – класс для хранения и работы со списком иконок. Хранит в себе данные о формате размеров иконки и холста для их отображения, функция для загрузки изображений, отработки нажатия на иконки.

Отображение иконок можно реализовать с помощью компонента ListBox, передавая в него изображения, использующиеся в качестве иконок.

Для того чтобы загрузить массив изображений из директории можно использовать следующий код:

```
using System.IO;

public void Load(string path)
{
    //путь до папки содержащей изображения
    string folder =
        System.IO.Path.GetDirectoryName(Process.GetCurrentProcess().MainModule.FileName) + path;

    //фильтр расширения изображения
    string filter = "*.png";

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

    foreach (string file in files)
    {
        //в file содержится путь до изображения с расширением .png
    }
}
```


Для создания иконки из загруженного изображения можно создать прямоугольный контейнер Rectangle:

```
public void CreateIcon(int iconWidth, int iconHeight, string imagePath)
{
    position = new Point(0, 0);

    name = System.IO.Path.GetFileNameWithoutExtension(imagePath);

    icon = new Rectangle();
    //установка цвета линии обводки и цвета заливки при помощи коллекции кистей
    icon.Stroke = Brushes.Black;
    ImageBrush ib = new ImageBrush();
    //позиция изображения будет указана как координаты левого верхнего угла
    //изображение будет растянуто по размерам прямоугольника, описанного вокруг фигуры
    ib.AlignmentX = AlignmentX.Left;
    ib.AlignmentY = AlignmentY.Top;

    //загрузка изображения и назначение кисти
    ib.ImageSource = new BitmapImage(new Uri(imagePath, UriKind.Absolute));

    icon.RenderTransform = new TranslateTransform(position.X, position.Y);

    icon.Fill = ib;
    //параметры выравнивания
    icon.HorizontalAlignment = HorizontalAlignment.Left;
    icon.VerticalAlignment = VerticalAlignment.Center;
    //размеры прямоугольника
    icon.Height = iconHeight;
    icon.Width = iconWidth;
}
```

Данный код может быть использован в качестве конструктора класса IIcon.

Для отображения иконки в окошке программы создайте Canvas в который можно добавить Rectangle в качестве дочернего элемента так же как были добавлены Line на Canvas в предыдущем задании. Для того чтобы отслеживать нажатия мыши следует добавить событие MouseDown к окну программы. Используя получение позиции от конкретного элемента WPF позволит получить координаты мыши в координатах выбранного элемента:

```
<Window x:Class="Editor.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Editor_v0_1"
        mc:Ignorable="d"
        Title="Editor_v0.1"           Height="423"           Width="825"           ResizeMode="NoResize"
        MouseDown="Window_MouseDown">
    <Grid>
    </Grid>
</Window>
```

Код для отслеживания нажатия на компонент:

```
private void Window_MouseDown(object sender, MouseButtonEventArgs e)
{
    //получение координат мыши в координатах объекта Canvas с именем scene
    Point mousePosition = Mouse.GetPosition(scene);
}
```

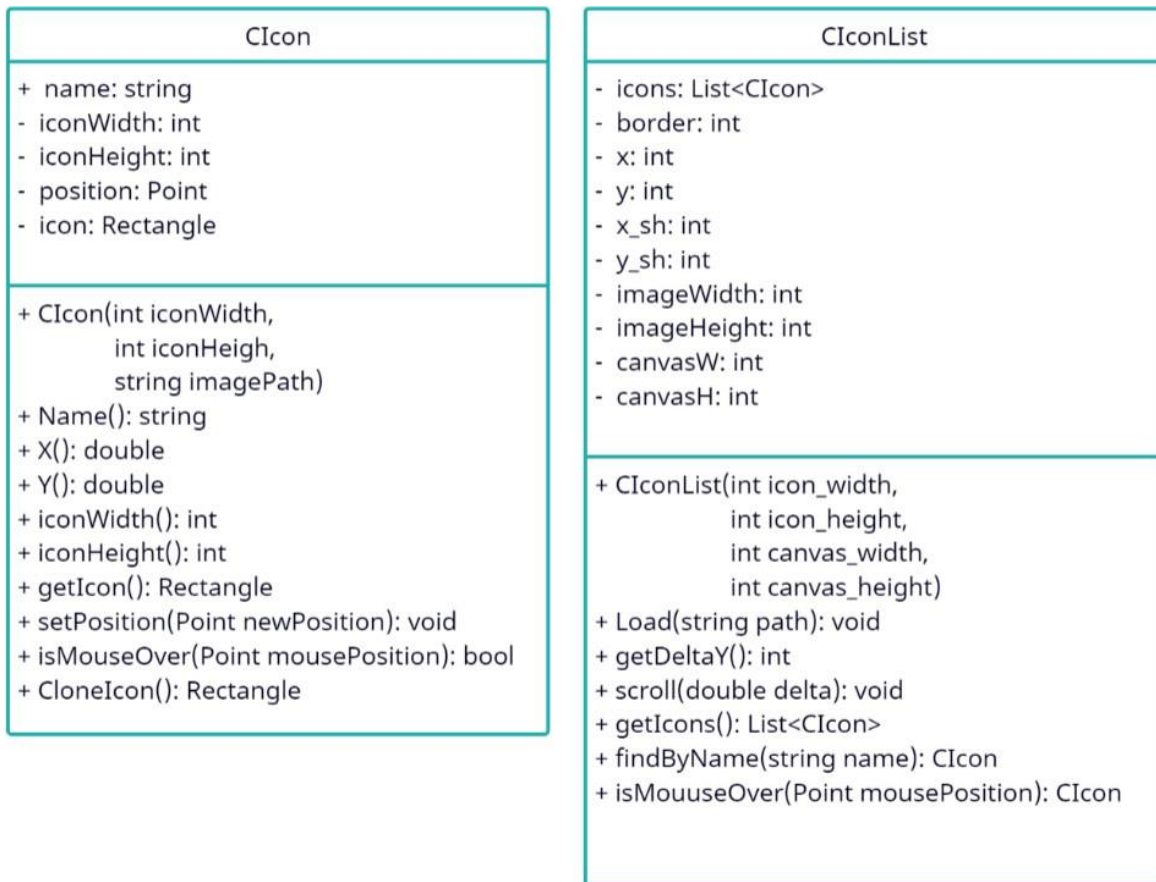


Рисунок 7 – UML-диаграммы классов работы с иконками

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;
}
```

Отчет:

Результат выполнения работы предоставить в виде:

- Архив с проектом (если размер архива больше 2 Мбайт, то рекомендуется загрузить проект на <https://github.com/> или на другое общедоступное хранилище и предоставить ссылку);
- Отчет по лабораторной работе в формате Microsoft Word, который содержит следующие разделы:
 1. титульный лист;
 2. задание на лабораторную работу;
 3. краткое описание разработанных программ и используемых алгоритмов со скриншотами выполнения;
 4. вывод о проделанной работе.

Список литературы:

- 1) Герберт Шилдт "С# 4.0: полное руководство"
- 2) Эндрю Троелсен "Язык программирования С# 5.0 и платформа .NET 4.5"
- 3) Полное руководство по языку программирования С# 7.0 и платформе .NET 4.7
<https://metanit.com/sharp/tutorial/>
- 4) Руководство по WPF <https://metanit.com/sharp/wpf/>
- 5) С# 5.0 и платформа .NET 4.5
http://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php
- 6) <https://github.com/Microsoft/WPF-Samples>