

## Практическая работа 4 Разработка модулей на основе спецификации

**Цель занятия:** Получить практический опыт разработки модулей на основе спецификации

### Перечень оборудования и программного обеспечения

Персональный компьютер  
Microsoft Office (Word, Visio)  
Microsoft Visual Studio 2015

### Краткие теоретические сведения

Программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова `class`:

```
class Person  
{  
  
}
```

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файл, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:

```
using System;  
  
namespace HelloApp  
{  
    class Person  
    {
```

```

}
class Program
{
    static void Main(string[] args)
    {

    }
}
}

```

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

```

using System;

namespace HelloApp
{
    class Person
    {
        public string name; // имя
        public int age = 18; // возраст

        public void GetInfo()
        {
            Console.WriteLine($"Имя: {name} Возраст: {age}");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person tom;
        }
    }
}

```

В данном случае класс Person представляет человека. Поле name хранит имя, а поле age - возраст человека. А метод GetInfo выводит все данные на консоль. Чтобы все данные были доступны вне класса Person переменные и метод определены с модификатором public. Поскольку поля фактически те же переменные, им можно присвоить начальные значения, как в случае выше, поле age инициализировано значением 18.

Так как класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе Main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение null. Поэтому вначале необходимо создать объект класса Person.

### **Конструкторы**

Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Person:

```
class Person
{
    public string name; // имя
    public int age;    // возраст

    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name} Возраст: {age}");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person();
        tom.GetInfo(); // Имя: Возраст: 0

        tom.name = "Tom";
        tom.age = 34;
        tom.GetInfo(); // Имя: Tom Возраст: 34
        Console.ReadKey();
    }
}
```

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен

участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";.

Консольный вывод данной программы:

Имя: Возраст: 0

Имя: Tom Возраст: 34

Можем определить свои конструкторы:

```
class Person
{
    public string name;
    public int age;

    public Person() { name = "Неизвестно"; age = 18; }    // 1 конструктор

    public Person(string n) { name = n; age = 18; }      // 2 конструктор

    public Person(string n, int a) { name = n; age = a; } // 3 конструктор

    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name} Возраст: {age}");
    }
}
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

```
static void Main(string[] args)
{
    Person tom = new Person();    // вызов 1-ого конструктора без параметров
    Person bob = new Person("Bob"); // вызов 2-ого конструктора с одним параметром
    Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами
}
```

```
bob.GetInfo();    // Имя: Bob Возраст: 18
tom.GetInfo();    // Имя: Неизвестно Возраст: 18
sam.GetInfo();    // Имя: Sam Возраст: 25
}
```

Консольный вывод данной программы:

Имя: Неизвестно Возраст: 18

Имя: Bob Возраст: 18

Имя: Sam Возраст: 25

При этом если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

Стоит отметить, что начиная с версии C# 9.0 мы можем сократить вызов конструктора, убрав из него название типа:

```
Person tom = new ();    // аналогично new Person();
Person bob = new ("Bob"); // аналогично new Person("Bob");
Person sam = new ("Sam", 25); // аналогично new Person("Sam", 25);
```

### Ключевое слово **this**

Ключевое слово **this** представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей `name` и `age`. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово **this**, передавая нужные значения для параметров:

```
class Person
{
    public string name;
    public int age;

    public Person() : this("Неизвестно")
    {
    }
    public Person(string name) : this(name, 18)
    {
    }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

```

public void GetInfo()
{
    Console.WriteLine($"Имя: {name} Возраст: {age}");
}
}

```

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

```

public Person(string name) : this(name, 18)
{
}

```

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

```

public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}

```

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово `this`. Так, в выражении `this.name = name`; первая часть `this.name` означает, что `name` - это поле текущего класса, а не название параметра `name`. Если бы у нас параметры и поля назывались по-разному, то использовать слово `this` было бы необязательно. Также через ключевое слово `this` можно обращаться к любому полю или методу.

### **Инициализаторы объектов**

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

```

Person tom = new Person { name = "Tom", age=31 };
tom.GetInfo(); // Имя: Tom Возраст: 31

```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.

Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

## Задания

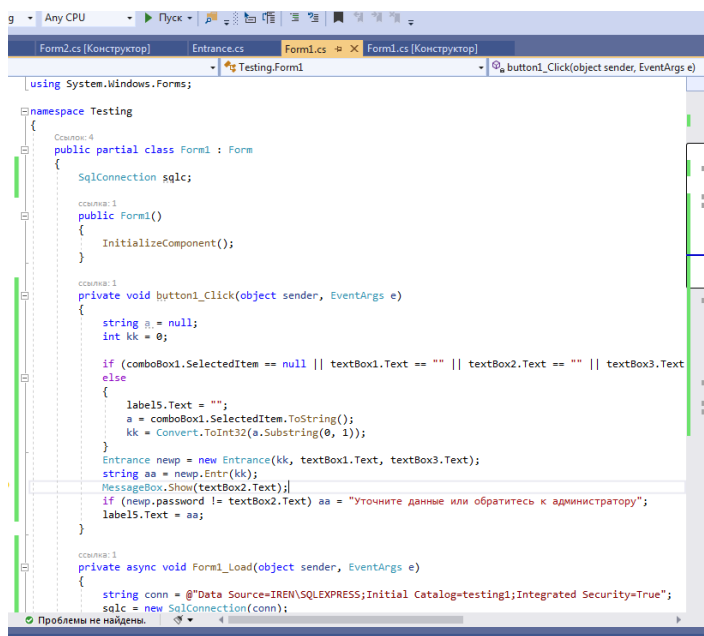
- 1 Изучить теоретические сведения и задание к работе
- 2 В соответствии с вариантом задания разработать функциональные модули приложения.

## Порядок выполнения работы

### Модуль авторизации пользователя

Для работы с приложением любой пользователь должен выполнить авторизацию, для чего выбрать категорию в comboBox1, вписать фамилию, имя и пароль. Эти данные обрабатываются в классе Entrance, где выполняется поиск в базе данных. Если пользователь использует приложение впервые, ему предлагается пройти регистрацию.

При загрузке Form1 происходит подключение к базе данных для формирования списка категорий пользователей. (Листинг)



```
using System.Windows.Forms;

namespace Testing
{
    Ссылка: 4
    public partial class Form1 : Form
    {
        SqlConnection sqlc;

        Ссылка: 1
        public Form1()
        {
            InitializeComponent();
        }

        Ссылка: 1
        private void button1_Click(object sender, EventArgs e)
        {
            string a = null;
            int kk = 0;

            if (comboBox1.SelectedItem == null || textBox1.Text == "" || textBox2.Text == "" || textBox3.Text
            else
            {
                label5.Text = "";
                a = comboBox1.SelectedItem.ToString();
                kk = Convert.ToInt32(a.Substring(0, 1));
            }
            Entrance newp = new Entrance(kk, textBox1.Text, textBox3.Text);
            string aa = newp.Entr(kk);
            MessageBox.Show(textBox2.Text);
            if (newp.password != textBox2.Text) aa = "Уточните данные или обратитесь к администратору";
            label5.Text = aa;
        }

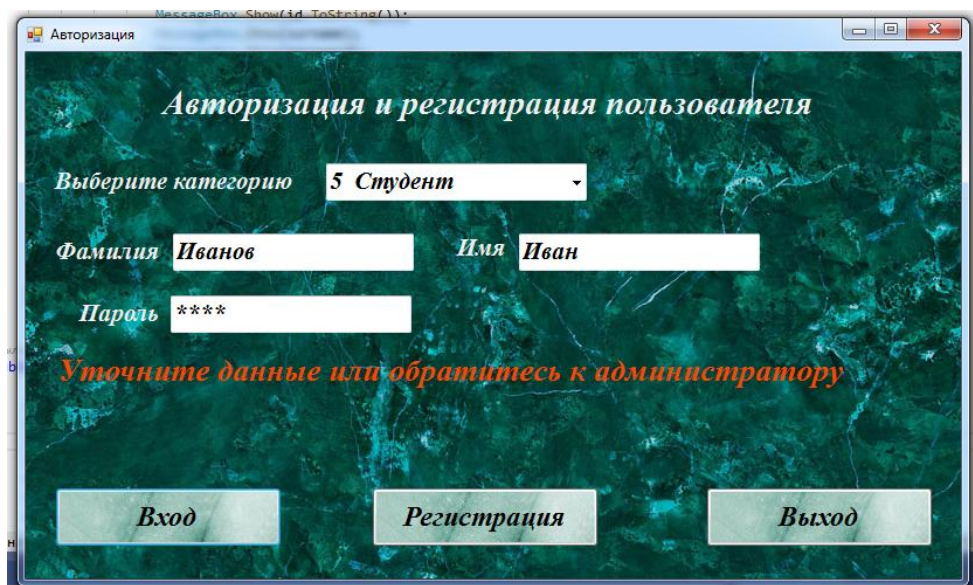
        Ссылка: 1
        private async void Form1_Load(object sender, EventArgs e)
        {
            string conn = @"Data Source=IREN\SQLEXPRESS;Initial Catalog=testing;Integrated Security=True";
            sqlc = new SqlConnection(conn);
        }
    }
}
```

В обработчике события нажатия кнопки Button1 создается экземпляр класса Entrance и происходит обработка данных пользователя. (Листинг)

```
namespace Testing
{
    class Entrance
    {
        public int id = 0, id_categ = 0, id_gr = 0;
        public string surname, name, patronymic, password;
        private SqlDataReader sqlr;

        public Entrance(int kk, string s, string n) // конструктор авторизации
        {
            if (kk != 6)
            {
                SqlCommand command;
                string conn = @"Data Source=IREM\SQLEXPRESS;Initial Catalog=testing1;Integrated Security=True";
                SqlConnection sqlc = new SqlConnection(conn);
                sqlc.OpenAsync();
                sqlr = null;
                MessageBox.Show(kk.ToString());
                //MessageBox.Show(s);
                //MessageBox.Show(n);
                if (kk == 5)
                {
                    command = new SqlCommand("SELECT * FROM [students] WHERE ([surname_stud] = @s) AND ([name_stud] = @n)", sqlc);
                    command.Parameters.AddWithValue("s", s);
                    command.Parameters.AddWithValue("n", n);
                }
                else
                {
                    command = new SqlCommand("SELECT * FROM [worker] WHERE ([surname_worker] = @s) AND ([name_worker] = @n)", sqlc);
                    command.Parameters.AddWithValue("s", s);
                    command.Parameters.AddWithValue("n", n);
                    command.Parameters.AddWithValue("kk", kk);
                }
            }
        }
    }
}
```

В результате в окне появляется сообщение или вызывается соответствующая форма для работы пользователя в зависимости от его категории.



...

**Содержание отчета**



1. Название работы
2. Цель работы
3. Перечень оборудования и программного обеспечения
4. Порядок выполнения
5. Листинг отлаженного модуля в соответствии с вариантом задания.
6. Вывод.

### **Варианты заданий**

Варианты заданий представлены в практической работе 1.

### **Используемая литература**

1 Рудаков А.В. Технология разработки программных продуктов: учеб. пособие для студ. учреждений сред. проф. образования – 11-е изд., стер. – М.: Издательский центр «Академия», 2017. – 208 с.

2 Федорова Г.Н. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности: учеб. пособие для студ. учреждений сред. проф. образования / Г.Н. Федорова.– М.: КУРС : ИНФРА – М, 2017. – 334 с

3 Федорова Г.Н. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности: Учебное пособие для СПО.- М.: КУРС, 2018. – 333 с.– ЭБС Знаниум

4 <http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx>.