Содержание

ВВЕДЕНИЕ **2**

АРХИТЕКТУРА ПРИЛОЖЕНИЯ **3**

ДОКУМЕНТИРОВАНИЕ ПРИЛОЖЕНИЯ **5**

ХЕШ БИБЛИОТЕКА (HASHBL) **6**

ПРОЕКТИРОВАНИЕ 6

КЛАССHASHING 7

КЛАССHASHMAP<TVALUE> 11

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС И ЛОГИКА ПРИЛОЖЕНИЯ (APPUI) **14**

ПРОЕКТИРОВАНИЕ 14

Класс LocalDataManager 17

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС 20

Тестирование приложения (AppUITests) 25

Тест-методы 25

РЕЗУЛЬТАТЫТЕСТИРОВАНИЯ 26

РАБОТА ПРИЛОЖЕНИЯ **28**

Введение

Хеш-функция— функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется **хешированием**.

Мы разработали приложение, включающее реализацию алгоритмов хеширования на основе хеш-функций, ориентированное на область обработки и сохранения данных пользователей. Наш проект доступен на GitHub.

Описание работы приложения

Пользователь вводит логин и пароль и выбирает одну из двух функций «Вход» и «Регистрация». В зависимости от этого программа должна выполнить соответствующую логику:

1) Для входа – проверка корректности введенных данных, хеширование данных, поиск и сравнение хешей данных с «базой данных». Если валидация прошла успешно, известить об этом пользователя и выполнить вход в личный кабинет, в котором будут отображаться его данные. Некоторые данные можно будет изменить, а значит также реализоваться соответствующую для этого логику.
2) Для регистрации – проверка корректности введенных данных, хеширование и запись нового пользователя в «базу данных».

Важные нюансы

Из-за ограниченного времени и знаний наша БД будет представлять собой каталог бинарных файлов.

Будем учитывать факт того, что БД может быть достаточно велика, чтобы не помещаться в оперативную память, а значит нужно спроектировать систему так, чтобы она обрабатывала неограниченные объемы данных.

Архитектура приложения

Основные критерии

Язык программирования:C#

Платформа:.NetCore

Пользовательский интерфейс будет реализован на WPF

Разделение приложения на модули (подсистемы)

Наше приложение целесообразно разделить на несколько модулей:

1) Библиотека, в которой будет содержаться логика работы с хеш-функциями и хеш-таблицой – HashBL.
2) Модуль, отвечающий за пользовательский интерфейс и прилегающий к нему функционал: разметка формы, получение данных с формы, обработка событий, хранение, извлечение и запись данных - AppUI.
3) Модуль, предназначенный для тестирования: проверки корректной работы при изменении приложения и оценки производительности некоторых функций - AppUITests.

Логику хеширования и производную от нее мы поместили в отдельную библиотеку затем, чтобы впоследствии функционал её можно было использовать в других приложения, путем добавления ссылки на сборку. Отсюда же следует, что, реализовывая библиотеку, мы должны абстрагироваться от предметной области логинов и паролей и для более гибкого использования добавить *шаблоны*.

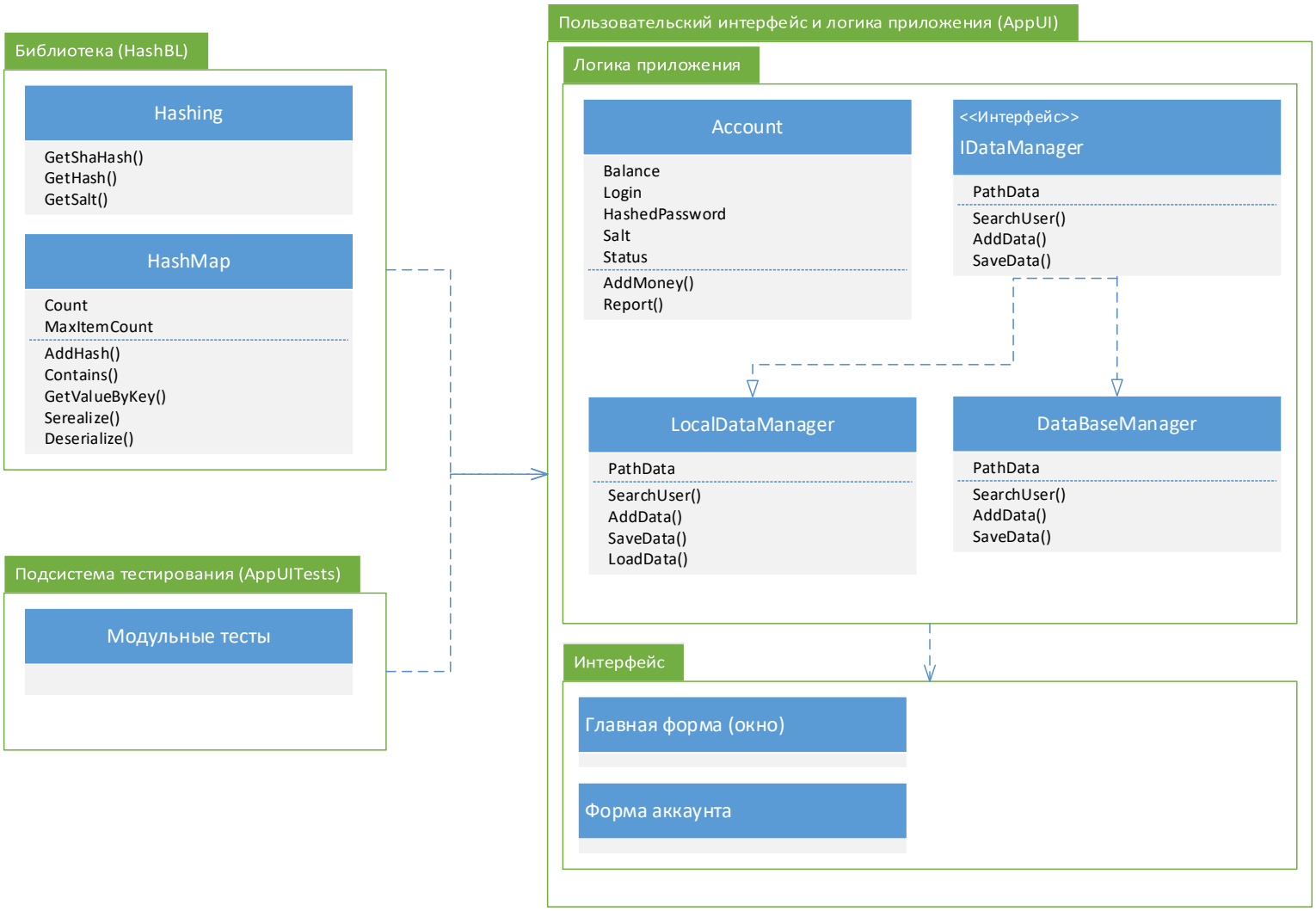


Рис. 1 Схема архитектуры

Документирование приложения

Так как разработка приложения шла совместно, использование интерфейсов можно считать оправданным (в модуле AppUI тем более, и позже объясним почему), а документирование – просто необходимым. Мы документировали приложение сразу в коде, используя *XML-комментарии*. Иногда мы выносили их в интерфейсы, а иногда делали это в классах. Это позволит не только отображать информацию об используемых классов/методов в среде разработке посредством IntelliSense, но и генерировать xml файл документации.

Хеш библиотека (HashBL)

Данная библиотека содержит два публичных класса: *Hashing*и *HashMap***,** а также интерфейс *IHashTable*.

Проектирование

Класс Hashing

Так как этот класс выступает в роли «функциональной коробочки», то нет смысла создавать экземпляры данного класса, а потому мы сделаем его статическим, это значит, что обращаться ко всем публичным методам, свойствам и полям мы можем без создания экземпляра этой абстракции.

Интерфейс должен содержать методы получения хеша и соли, скрывая внутренние детали реализации конкретных алгоритмов, т.е инкапсулируя.

Класс HashMap

Этот класс представляет структуру данных – хеш-таблицу. Хранение данных организовано с помощью словаря, ключом которого является ulong переменная – это будет хеш логина, а значением тип TValue – шаблон, придающий гибкость использования данного класса. В дальнейшей разработке мы решили сделать специальный тип данных для аккаунтов пользователей, объявив Account класс, о котором расскажем чуть позже.

Можно сказать, что данный класс является оболочкой над структурой данных «словарь». Функционал данной структуры довольно широк, а нам нет необходимости пользоваться большей её частью. Помня о *главном техническом императиве,* мы скроем лишний функционал и добавим свои необходимые методы для сериализации и десериализации.

Как говорилось во введении, объем данных может быть достаточно большим, поэтому мы установим некоторый оптимальный максимум в структуре, и при ее переполнении будет сохранять данные в файл. Лучшим решением будет скрыть данные о максимальном кол-ве элементов, определив их полем

MaxItemCount, значение которого при необходимости можно будет изменить в конструкторе класса.

Интерфейс класса должен содержать необходимые методы для обработки коллекции, её сериализации и десериализации.

Класс Hashing

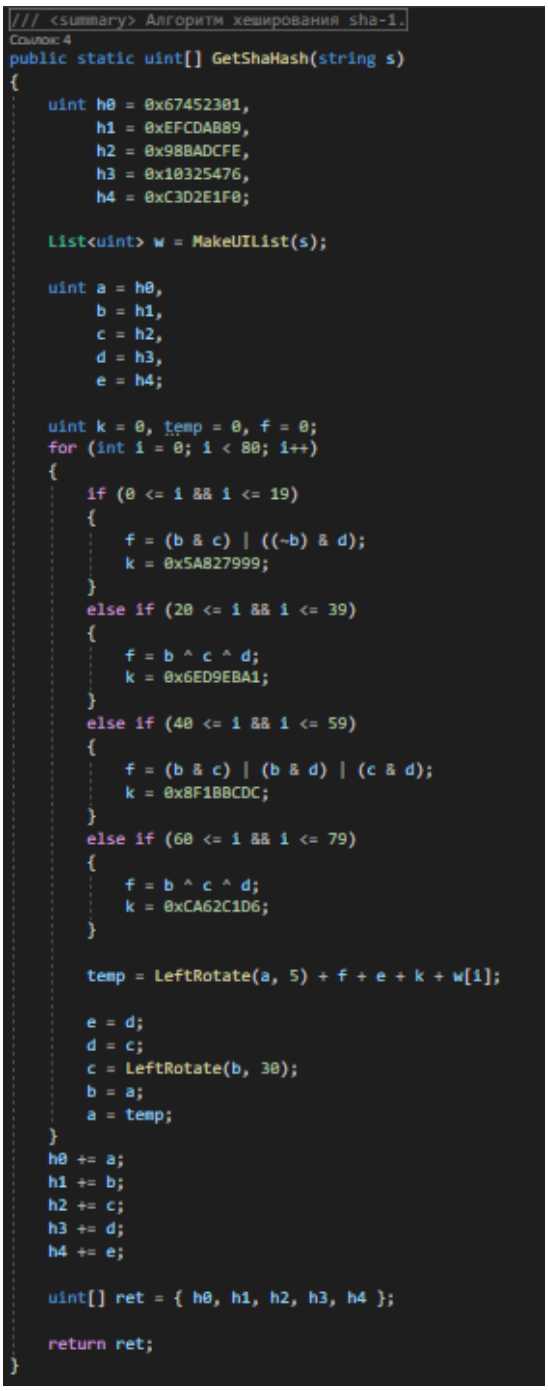
В этом классе мы реализовали несколько алгоритмов хеширования, а также генерацию соли.

Публичные методы

Алгоритм хеширования SHA-1

Метод *GetShaHash*

Реализует алгоритм хеширования SHA-1, который преобразует строку в 512 битноесообщение. Входящей строкой является пароль пользователя сканкотанированный с солью, которая генерируется методом, описанным ниже. Сам алгоритм сначала преобразует входящую строку в битовый вид, после в конец строки добавляются биты до тех пор, пока длина не будет равняться 512 битам. Потом последние 64 бита заменяются на двоичное представление длины входящей строки. После с помощью рекуррентной формулы длина битового сообщения увеличивается до 2560 бит. Затем с помощью битовых операций в цикле и пяти заданных констант мы получаем хэш значение.



```
/// <summary> Алгоритм хеширования sha-1.
Ссылок: 4
public static uint[] GetShaHash(string s)
{
    uint h0 = 0x67452301,
         h1 = 0xEFCDAB89,
         h2 = 0x98BADCFE,
         h3 = 0x10325476,
         h4 = 0xC3D2E1F0;

    List<uint> w = MakeUIList(s);

    uint a = h0,
         b = h1,
         c = h2,
         d = h3,
         e = h4;

    uint k = 0, temp = 0, f = 0;
    for (int i = 0; i < 80; i++)
    {
        if (0 <= i && i <= 19)
        {
            f = (b & c) | ((~b) & d);
            k = 0x5A827999;
        }
        else if (20 <= i && i <= 39)
        {
            f = b ^ c ^ d;
            k = 0x6ED9EBA1;
        }
        else if (40 <= i && i <= 59)
        {
            f = (b & c) | (b & d) | (c & d);
            k = 0x8F1BBCDC;
        }
        else if (60 <= i && i <= 79)
        {
            f = b ^ c ^ d;
            k = 0xCA62C1D6;
        }

        temp = LeftRotate(a, 5) + f + e + k + w[i];

        e = d;
        d = c;
        c = LeftRotate(b, 30);
        b = a;
        a = temp;
    }
    h0 += a;
    h1 += b;
    h2 += c;
    h3 += d;
    h4 += e;

    uint[] ret = { h0, h1, h2, h3, h4 };

    return ret;
}
```

рис.2.1 Алгоритм SHA-1

Алгоритм хеширования через простое число в степени

Метод *GetHash*

Реализует алгоритм простого хеширования с использованием простого числа в степени и кода символа. Данный метод нужен только для хеширования логинов, а получившийся хэш будет использоваться в качестве ключа в структуре.

```
/// <summary> Алгоритм хеширования через простое число в степени.
Ссылок: 5
public static ulong GetHash(string s)
{
    ulong answer = 0;
    short primeNumber = 67;

    for (int i = 0; i < s.Length; i++)
    {
        answer += (ulong)(s[i] - '0' + 1) * (ulong)Math.Pow(primeNumber, i + 1);
    }

    return answer;
}
```

рис.2.2 Второй алгоритм хеширования

Алгоритм получения соли

Метод *GetSalt*

Генерирует соль, т. е. строку фиксированной длины со случайными символами. Эта строка по ходу алгоритма прибавится к паролю перед его хешированием. Соль необходима для усложнения определения прообраза хэш-функции.

```
/// <summary> Генерация соли.
Ссылок: 2
public static string GetSalt(int n = 12)
{
    string ret = "";

    Random GenChar = new Random();
    Random GenReg = new Random();

    for (int i = 0; i < n; i++)
    {
        char c;
        if (GenReg.Next() % 2 == 0)
            c = 'a';
        else
            c = 'A';

        c = Convert.ToChar(c + GenChar.Next() % 26);

        ret += c;
    }

    return ret;
}
```

рис.2.3 Получение соли

Закрытые методы

Вспомогательные методы для хеширования SHA-1

Методы *LeftRotate* и *MakeUIList*

Первый метод реализует циклический сдвиг на определенное количество бит. Второй метод преобразует строку, которая является паролем, в список из беззнаковых интеджеров (*uint*) длиной в 80 символов. Этот список будет нужен по ходу алгоритма хеширования пароля.

```
/// <summary> Циклический сдвиг влево.
Ссылок: 3
private static uint LeftRotate(uint a, int b)
{
    b %= 32;
    if (b == 0)
        return a;

    uint x = a >> (32 - b);
    a = a << b;
    a = a | x;
    return a;
}
```

```
private static List<uint> MakeUIList(string s)
{
    List<uint> ret = new List<uint>();
    uint tmp = 0;

    for (int i = 0; i < s.Length; i++)
    {
        tmp = tmp << 8;
        tmp += (Convert.ToUInt32(s[i]));

        if (i % 4 == 3)
        {
            ret.Add(tmp);
            tmp = 0;
        }
    }
    ret.Add(tmp);

    tmp = ret[ret.Count - 1];

    if (tmp != 0)
    {
        int step = 0;
        while ((tmp & 0xff000000) == 0)
        {
            tmp = tmp << 8;
            step += 8;
        }
        uint one = 1;
        one = one << (step - 1);
        tmp = tmp + one;
    }
    else
    {
        tmp = 1;
        tmp = tmp << 31;
    }

    ret[ret.Count - 1] = tmp;

    while (ret.Count != 15)
    {
        ret.Add(0);
    }

    ret.Add(Convert.ToUInt32(s.Length * 8));

    for (int i = 16; i < 80; i++)
    {
        tmp = ret[i - 3] ^ ret[i - 8] ^ ret[i - 14] ^ ret[i - 16];
        tmp = LeftRotate(tmp, 1);
        ret.Add(tmp);
        tmp = 0;
    }

    return ret;
}
```

рис.2.4 - 2.5 Вспомогательные методы

Класс HashMap<TValue>

Публичные методы

Методы сохранения данных в файл

Сохранение самой хеш-таблицы было решено сделать с помощью сериализации (и десериализации соответственно). За это отвечают методы *Serialize(string path)* и *Deserialize(string path)*.

```
Ссылок: 4
public void Serealize(string path)
{
    using (FileStream file = new FileStream(path, FileMode.Create))
    {
        format.Serialize(file, hashMap);
        hashMap = new Dictionary<ulong, TValue>(MaxItemCount);
    }
}

Ссылок: 4
public void Deserialize(string path)
{
    if (File.Exists(path))
    {
        using (FileStream file = new FileStream(path, FileMode.Open))
        {
            hashMap = (Dictionary<ulong, TValue>)format.Deserialize(file);
        }
    }
    else
        throw new Exception("Такого файла не существует.");
}
```

рис.2.6 Сериализация и десериализация

Добавление значений в структуру

Добавление данных определено методом *AddHash(ulonghash, TValuevalue).* Данный метод также должен проверять существование такого хеша, а также следить за тем, чтобы не произошло переполнение структуры данных.

```
Ссылок: 4
public void AddHash(ulong loginHash, TValue value)
{
    if (Contains(loginHash))
        throw new ArgumentException("Такой логин уже есть");

    else if(Count < MaxItemCount)
        hashMap[loginHash] = value;
    else
        throw new OverflowException("Достигнуто максимальное кол-во элементов");
}
```

рис.2.7 Добавление в структуру данных

Поиск

Поиск определен методом *Contains(ulongkey)*. Если заданный ключ имеется в коллекции, то возвращается *True*, иначе возвращается значение *False*.

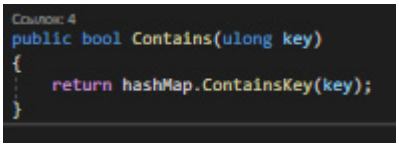


рис.2.8 Метод *Contains(ulong key)*

Получение значения по ключу

За данную операцию отвечает метод *GetValueByKey(ulongkey).* Для защиты от ошибок нам все равно необходимо проверить входные данные, т.е. определить, есть ли в коллекции вводимый ключ. Если заданный ключ имеется в коллекции, то возвращается значение, привязанное к этому ключу, иначе возвращается значение *default* для типа *TValue*.

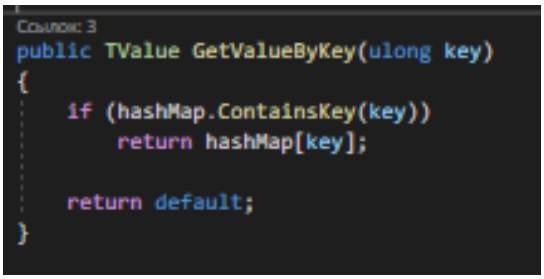


рис.2.8 Метод *GetValueByKey(ulong key)*

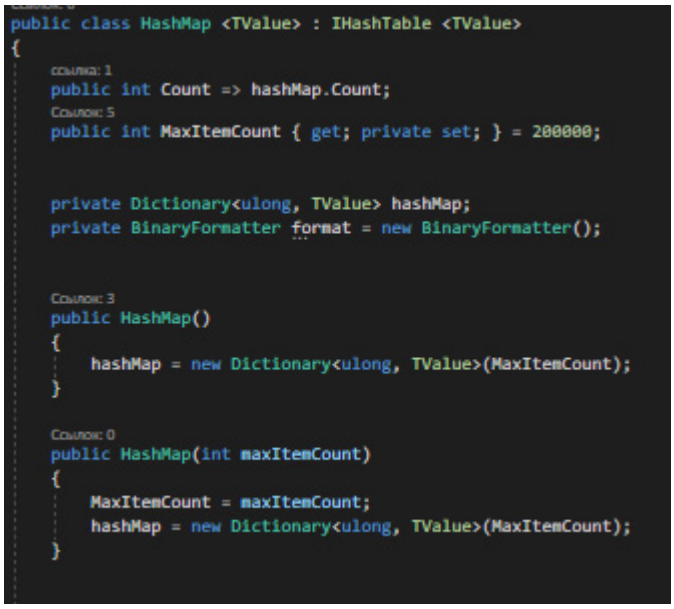
Свойства

Count

Возвращает кол-во элементов в коллекции. Установить значение невозможно, отсутствует аксессор set.

MaxItemCount

Возвращает максимально возможное кол-во элементов. Установить значение можно только внутри этого класса, например в конструкторе, так как аксессор set является приватным.



```
public class HashMap <TValue> : IHashTable <TValue>
{
    ссылка: 1
    public int Count => hashMap.Count;
    Ссылок: 5
    public int MaxItemCount { get; private set; } = 200000;


    private Dictionary<ulong, TValue> hashMap;
    private BinaryFormatter format = new BinaryFormatter();


    Ссылок: 3
    public HashMap()
    {
        hashMap = new Dictionary<ulong, TValue>(MaxItemCount);
    }

    Ссылок: 0
    public HashMap(int maxItemCount)
    {
        MaxItemCount = maxItemCount;
        hashMap = new Dictionary<ulong, TValue>(MaxItemCount);
    }
```

рис.2.9 Свойства *HashMap*

Пользовательский интерфейс и логика приложения (AppUI)

Данная подсистема содержит разметку и обработку пользовательского интерфейса: *MainWindow* и *AccountPage*. Публичные классы: *Account, LocalDataManager, DataBaseManager, User (для тестирования),* а также интерфейсы *IAccount* и *IDataManager.*

Проектирование

Класс Account

Объекты этого класса будут представлять пользователей системы. Класс имеет *[Serializable]* атрибут, так как его объекты будут храниться в структуре данных в качестве *TValue* значения.

Интерфейс должен содержать методы для обработки, доступа и изменения данных.

Класс LocalDataManager

Этот класс реализует обработку «локальной базы данных», имплементируя интерфейс *IDataManager.* Мы уже говорили, что сделали хранение данных в виде бинарных файлов. Однако думая о масштабируемости и практике, мы понимаем, что данному типу приложения уместно сделать настоящую базу данных, которая возможно будет располагаться и не локально. Поэтому при проектировании был создан интерфейс управления данными (*IDataManager*), имплементируя который можно распределить управление данных на локальное (*LocalDataManager*) и глобальное (*DataBaseManager*). Мы не будем напрямую обращаться к классам *LocalDataManager* и *DataBaseManager*, вместо этого создадим *IDataManager* «контейнер», это позволит сильно сократить кол-во изменений в коде - одной строки будет достаточно, причем сделать это можно даже программно.

Интерфейс включает в себя сохранение и добавление данных, путь к этим данным и поиск.

Класс DataBaseManager

В будущем в этом классе можно реализовать работу с настоящей базой данных.

Класс User

Данный класс нужен для программной генерации пользователей и используется в модуле для тестов, чтобы сократить код тест-методов.

Класс Account

Публичные методы

Зачисление денег на баланс

Метод *AddMoney*

Увеличивает баланс пользователя.

Отправка отчета

Метод *Report*

Потенциально данный метод будет формировать отчёт об информации аккаунта пользователя.

Свойства

Status

Возвращает статус пользователя, определяющийся значением перечисления. При создании объекта класса Acccount, значение устанавливается на Normal, в будущем при помощи специальных методов можно будет менять статус пользователя и тем самым ограничивать его функционал.

Salt

Возвращает соль пользователя, значение устанавливается при создании объекта класса Acccount и нужно при проверке пароля пользователя.

Balance

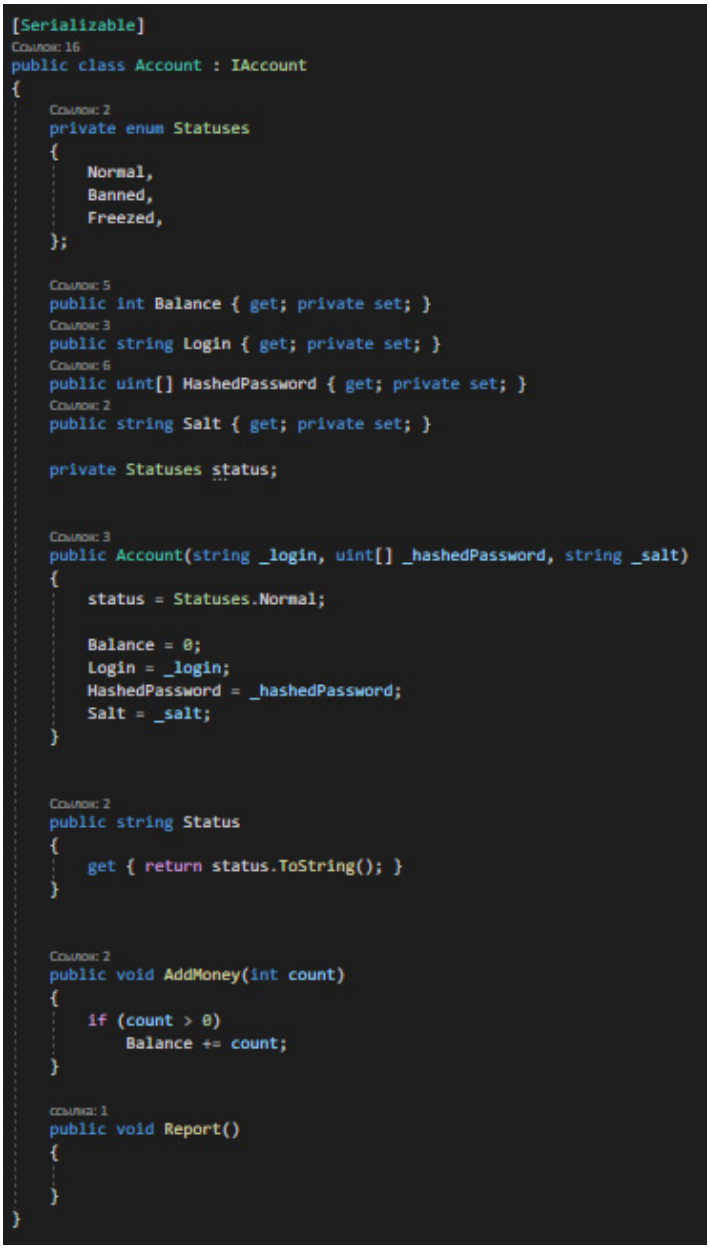
Возвращает количество денег пользователя, при создании объекта класса Acccount устанавливается на 0 и увеличивается при помощи метода AddMoney.

Login

Возвращает логин пользователя, значение устанавливается при создании объекта класса Acccount.

HashedPassword

Возвращает хэш пароля пользователя, значение устанавливается при создании объекта класса Acccount.



```
[Serializable]
Ссылок: 16
public class Account : IAccount
{
    Ссылок: 2
    private enum Statuses
    {
        Normal,
        Banned,
        Freezed,
    };

    Ссылок: 5
    public int Balance { get; private set; }
    Ссылок: 3
    public string Login { get; private set; }
    Ссылок: 6
    public uint[] HashedPassword { get; private set; }
    Ссылок: 2
    public string Salt { get; private set; }

    private Statuses status;


    Ссылок: 3
    public Account(string _login, uint[] _hashedPassword, string _salt)
    {
        status = Statuses.Normal;

        Balance = 0;
        Login = _login;
        HashedPassword = _hashedPassword;
        Salt = _salt;
    }


    Ссылок: 2
    public string Status
    {
        get { return status.ToString(); }
    }


    Ссылок: 2
    public void AddMoney(int count)
    {
        if (count > 0)
            Balance += count;
    }

    ссылка: 1
    public void Report()
    {

    }
}
```

рис.3Класс *Account*

Класс LocalDataManager

Публичные методы

Поиск пользователя

Метод *SearchUser*

Осуществляет поиск аккаунта пользователя (в оперативной памяти и всем файлам базы данных) по ключу, получаемому методом хеширования логина. Значение передается выходному параметру с модификатором out. Сама же функция возвращает true если аккаунт был найден и false если нет.

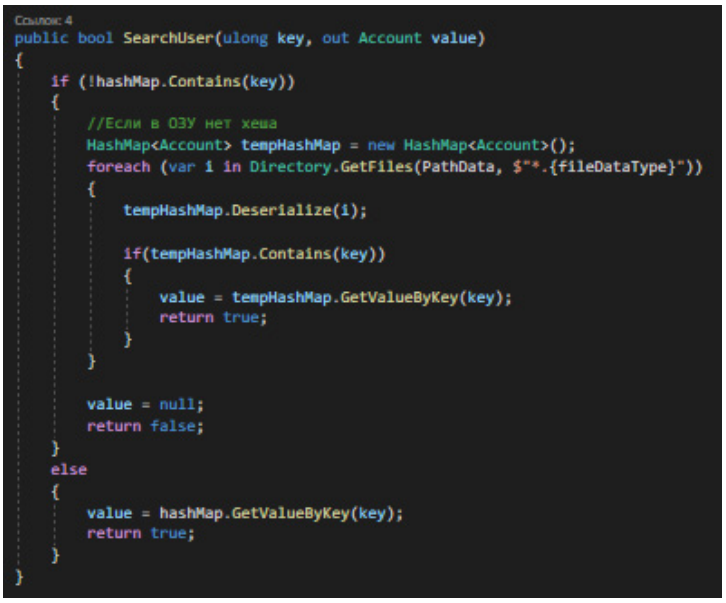


рис.3.1 Поиск пользователей в локальной «базе данных»

Загрузка данных в оперативную память

Метод *LoadData*

Загружает в ОЗУ последний созданный файл базы данных.

Сохранение данных

Метод *SaveData*

Сериализует данные вызовом метода *Serialize* объекта структуры данных *HashMap.*

Добавление данных

Метод *AddData*

Добавляет в базу данных нового пользователя, либо создаёт новый файл и добавляет аккаун в него.

```
/// <summary> Загружает данные в ОЗУ.
ссылка: 1
public void LoadData(string folderPath, string fileType = "data")
{
    PathData = folderPath;
    fileDataType = fileType;

    // Создаем папку для хранения данных, если таковая отсутствует
    if (!Directory.Exists(PathData))
        Directory.CreateDirectory(PathData);

    string[] fileNames = Directory.GetFiles(PathData, $"*.{fileDataType}");
    if (fileNames.Length > 0)
    {
        hashMap.Deserialize(fileNames[fileNames.Length - 1]);
        nowFileName = $@"{PathData}\file{fileNames.Length - 1}.{fileDataType}";
    }
    else
        nowFileName = $@"{PathData}\file{fileNames.Length}.{fileDataType}";
}

Ссылок: 3
public void SaveData()
{
    hashMap.Serealize(nowFileName);
}

Ссылок: 3
public void AddData(string login, string password)
{
    string salt = Hashing.GetSalt();
    try
    {
        hashMap.AddHash(Hashing.GetHash(login), new Account(login, Hashing.GetShaHash(password + salt), salt));
    }
    catch (OverflowException)
    {
        hashMap.Serealize(nowFileName);
        nowFileName = $@"{PathData}\file{Directory.GetFiles(PathData, $"*.{fileDataType}").Length}.{fileDataType}";
        hashMap.AddHash(Hashing.GetHash(login), new Account(login, Hashing.GetShaHash(password + salt), salt));
        throw new OverflowException();
    }
}
```

рис.3.2Методы *LocalDataMager'a*

Свойства

PathData

Возвращает местоположения данных. Для этого менеджера - имя папки, в которой хранятся бинарные файлы.

рис.3.3Свойство *PathData*

Пользовательский интерфейс.

События пользовательского интерфейса

Кнопка «Войти» (Располагается на главной форме)

Метод btnSignInClick

Срабатывает при нажатии на кнопку «Войти». Сначала происходит проверка логина и пароля на корректность, затем определяется, есть ли такой пользователь в базе данных и также сравниваются хэшы паролей и если всё совпадает, то приложение разрешает вход - пользователь переходит на форму аккаунта.

```
/// <summary> Обработка клика мыши по кнопке "Войти".
ссылка: 1
private void btnSignInClick(object sender, RoutedEventArgs e)
{
    MyMessageBox.Text = "";
    GetTextFromTextBox(out login, out password);

    if (!CheckData(login, password))
        return;

    ulong loginHash = Hashing.GetHash(login);
    Account accountValue = null;

    if(dataManager.SearchUser(loginHash, out accountValue))
    {
        MyMessageBox.Text = "";
        uint[] hashUser = Hashing.GetShaHash(password + accountValue.Salt);

        bool check = true;

        for (int i = 0; i < hashUser.Length && i < accountValue.HashedPassword.Length; i++)
        {
            if (hashUser[i] != accountValue.HashedPassword[i])
            {
                check = false;
                break;
            }
        }

        if (check)
        {
            DisplayMessage("Доступ разрешен.", "", MessageBoxImage.Information);
            string s = accountValue.Status;
            MainFrame.Content = new AccountPage(MainFrame, accountValue);
        }

        else
            DisplayError(ErrorType.UnMatching);
    }
    else
        DisplayError(ErrorType.UnMatching);
}
```

рис.4Кнопка «Войти»

Кнопка «Зарегистрироваться» (Располагается на главной форме)

Метод btnReistrationClick

Срабатывает при нажатии на кнопку «Зарегистрироваться». Сначала также происходит проверка логина и пароля на корректность, затем определяется есть ли такой пользователь в базе данных и если такого нет, то происходит добавление нового пользователя в «базу данных».

```
/// <summary> Обработка клика мыши по кнопке "Зарегистрироваться".
ссылка: 1
private void btnReistrationClick(object sender, RoutedEventArgs e)
{
    MyMessageBox.Text = "";
    GetTextFromTextBox(out login, out password);

    if (!CheckData(login, password))
        return;

    ulong loginHash = Hashing.GetHash(login);
    Account accountValue;

    if (dataManager.SearchUser(loginHash, out accountValue))
        DisplayError(ErrorType.LoginExsist);
    else
    {
        MyMessageBox.Text = "";
        try
        {
            dataManager.AddData(login, password);
        }
        catch (OverflowException)
        {
            DisplayMessage("Случилось переполнение хеш-таблицы.\n" +
                        "Программа сохранила данные в файл и записала нового пользователя.", "Внимание", MessageBoxImage.Information);
        }
    }
}
```

рис.4.1Кнопка «Зарегистрироваться»

Кнопка «Зачислить средства» (Располагается на форме аккаунта)

Метод addMoneyBtn_Click

Проверяет вводимые данные и зачисляет деньги на баланс.

```
ссылка: 1
private void addMoneyBtn_Click(object sender, RoutedEventArgs e)
{
    string money_s = moneyBox.Text;
    if (money_s.Length != 0)
    {
        for (int i = 0;i < money_s.Length;i++)
            if (!('0' <= money_s[i] && money_s[i] <= '9'))
            {
                moneyBox.Clear();
                return;
            }

        int money = int.Parse(money_s);

        myAccount.AddMoney(money);
        BalanceCountLabel.Content = myAccount.Balance.ToString();

        moneyBox.Clear();
    }
}
```

рис.4.2Кнопка «Зачислить средства»

Кнопка «Выйти из аккаунта» (Располагается на форме аккаунта)

Метод exitBtn_Click

Возвращает пользователя на главную форму.

```
ссылка: 1
private void exitBtn_Click(object sender, RoutedEventArgs e)
{
    MainFrame.Content = null;
}
```

рис.4.3 Кнопка «Выйти из аккаунта»

Проверка входных данных и предотвращение ошибок

Мы реализовали несколько методов по получению и проверке данных и предотвращении ошибок. Подробно мы останавливаться на них не будем, но ознакомиться с ними можно на рисунках 4.4 и 4.5.

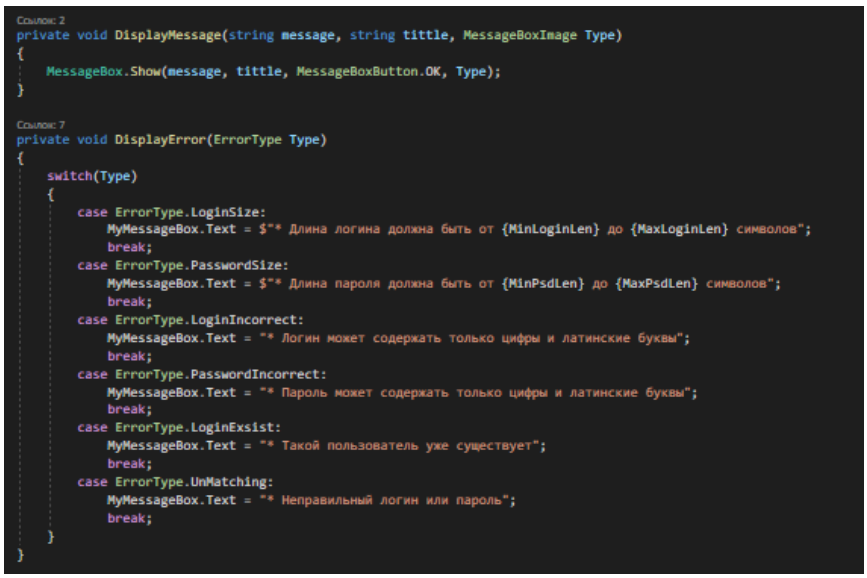
```
/// <summary> Проверка корректности ввода
Ссылок: 2
private bool CheckData(string login, string psd)
{
    if (!(MinLoginLen <= login.Length && login.Length <= MaxLoginLen))
    {
        DisplayError(ErrorType.LoginSize);
        return false;
    }

    for (int i = 0; i < login.Length; i++)
        if (!('a' <= login[i] && login[i] <= 'z') &&
            !('A' <= login[i] && login[i] <= 'Z') &&
            !('0' <= login[i] && login[i] <= '9'))
        {
            DisplayError( ErrorType.LoginIncorrect);
            return false;
        }

    if (!(MinPsdLen <= password.Length && password.Length <= MaxPsdLen))
    {
        DisplayError(ErrorType.PasswordSize);
        return false;
    }

    for (int i = 0; i < password.Length; i++)
        if (!('a' <= password[i] && password[i] <= 'z') &&
            !('A' <= password[i] && password[i] <= 'Z') &&
            !('0' <= password[i] && password[i] <= '9'))
        {
            DisplayError(ErrorType.PasswordIncorrect);
            return false;
        }

    return true;
}
```



```
Ссылок: 2
private void DisplayMessage(string message, string tittle, MessageBoxImage Type)
{
    MessageBox.Show(message, tittle, MessageBoxButton.OK, Type);
}

Ссылок: 7
private void DisplayError(ErrorType Type)
{
    switch(Type)
    {
        case ErrorType.LoginSize:
            MyMessageBox.Text = $"* Длина логина должна быть от {MinLoginLen} до {MaxLoginLen} символов";
            break;
        case ErrorType.PasswordSize:
            MyMessageBox.Text = $"* Длина пароля должна быть от {MinPsdLen} до {MaxPsdLen} символов";
            break;
        case ErrorType.LoginIncorrect:
            MyMessageBox.Text = "* Логин может содержать только цифры и латинские буквы";
            break;
        case ErrorType.PasswordIncorrect:
            MyMessageBox.Text = "* Пароль может содержать только цифры и латинские буквы";
            break;
        case ErrorType.LoginExsist:
            MyMessageBox.Text = "* Такой пользователь уже существует";
            break;
        case ErrorType.UnMatching:
            MyMessageBox.Text = "* Неправильный логин или пароль";
            break;
    }
}
```

рис 4.4–4.5 Методы для проверки входных данных и обработки ошибок.

Тестирование приложения (AppUITests)

Данная подсистема содержит несколько тест-методов, суть которых заключается в проверке корректности сохранения данных, а также учет производительности.

Тест-методы

Главная логика тест-метода

```
Ссылок: 3
void m(int count)
{
    //arrange
    var hashTable1 = User.HashUser(count);

    var dict1 = hashTable1.GetHashDict;

    //act
    hashTable1.Deserialize("HashData/file1.data");
    var dict2 = hashTable1.GetHashDict;

    //assert
    foreach (var key in dict1.Keys)
    {
        if (dict2.ContainsKey(key))
        {
            for (int i = 0; i < dict2[key].HashedPassword.Length; i++)
            {
                Assert.AreEqual(dict2[key].HashedPassword[i], dict1[key].HashedPassword[i]);
            }
        }
        else
            Assert.Fail();
    }

    //Assert.AreEqual(answer, answer);
}
```

рис.5 Логика тест метода

Тест методы

```
[TestMethod()]
Ссылок: 0
public void HashUser_Test10k()
{
    m(10000);
}

[TestMethod()]
Ссылок: 0
public void HashUser_Test60k()
{
    m(60000);
}

[TestMethod()]
Ссылок: 0
public void HashUser_Test180k()
{
    m(180000);
}
```

рис.5.1 Тест методы

Результаты тестирования

Из результатов тестирования видно, что на процесс создания пользователей (генерация логинов, солей и хешей), сериализация, десериализация и сравнение занимает 0.3, 1.7, 5.7 секунд для 10, 60, 180 тысяч пользователей соответственно.

Объем файлов с данными для 10к занимает 1.1 МБайт, для 60к–6.4 МБайт, 180к – 19.2 МБайт.

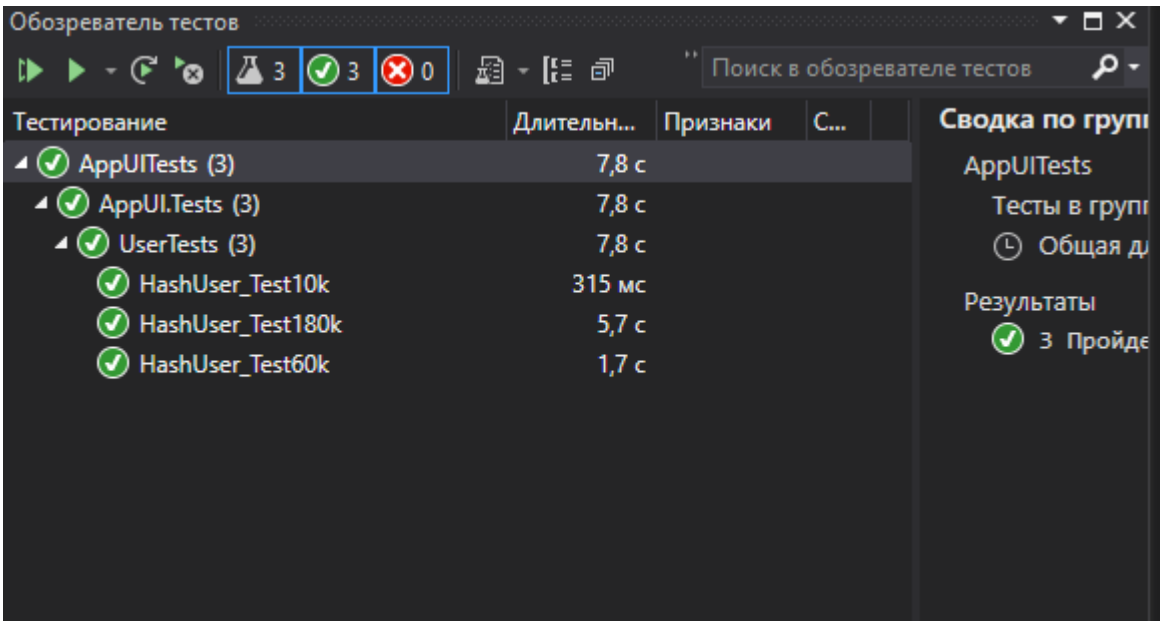
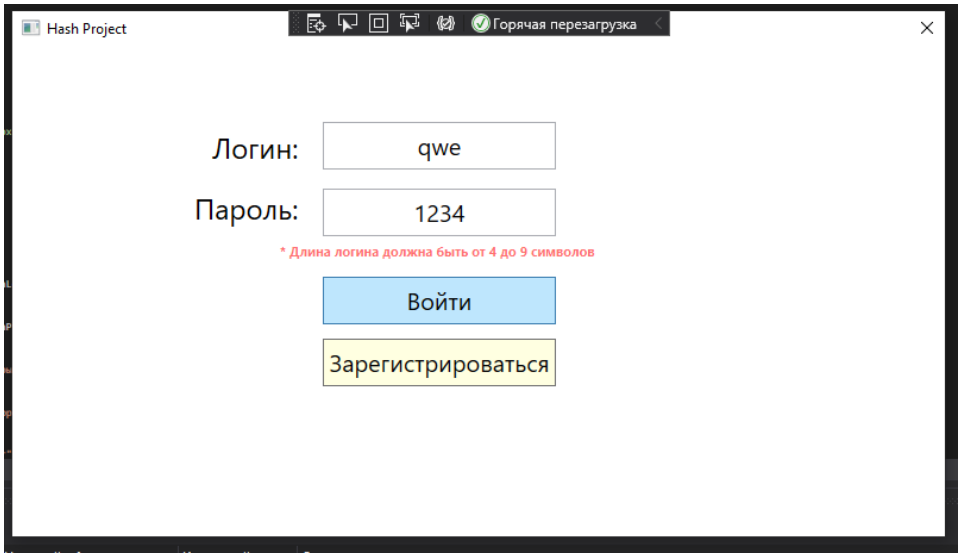
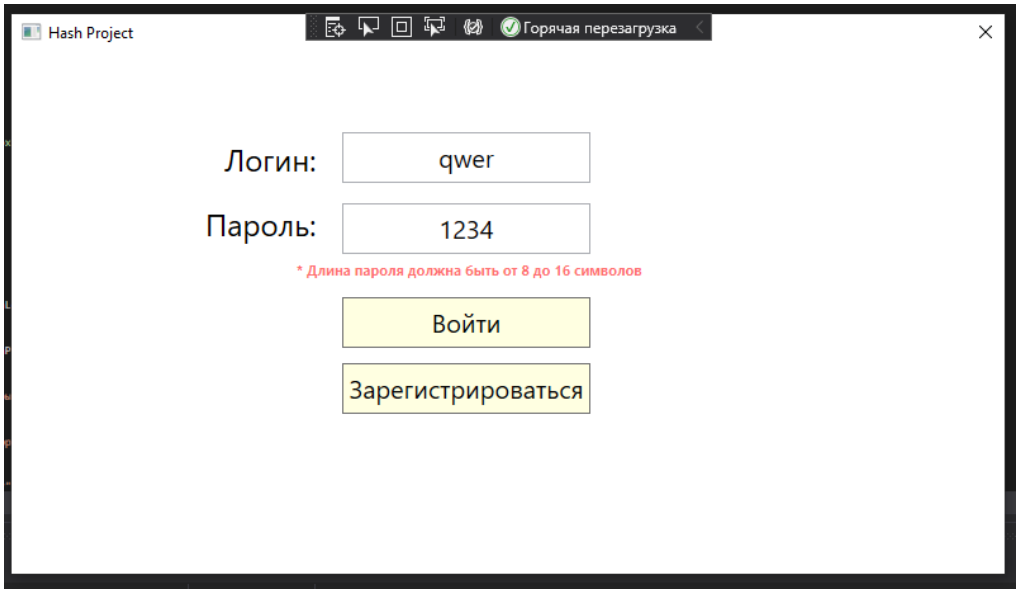


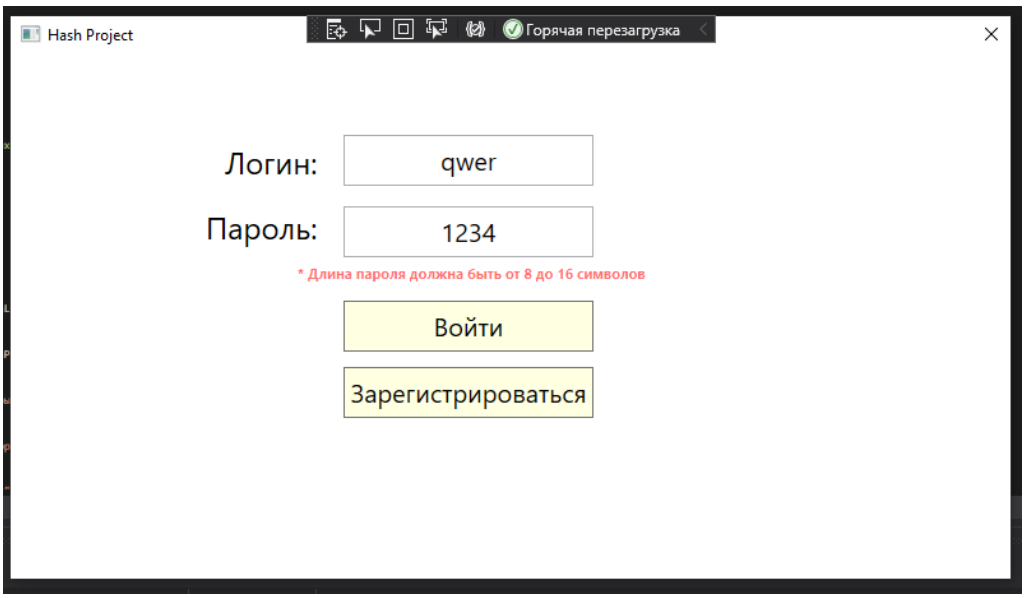
рис.5.2 Результаты тестирования

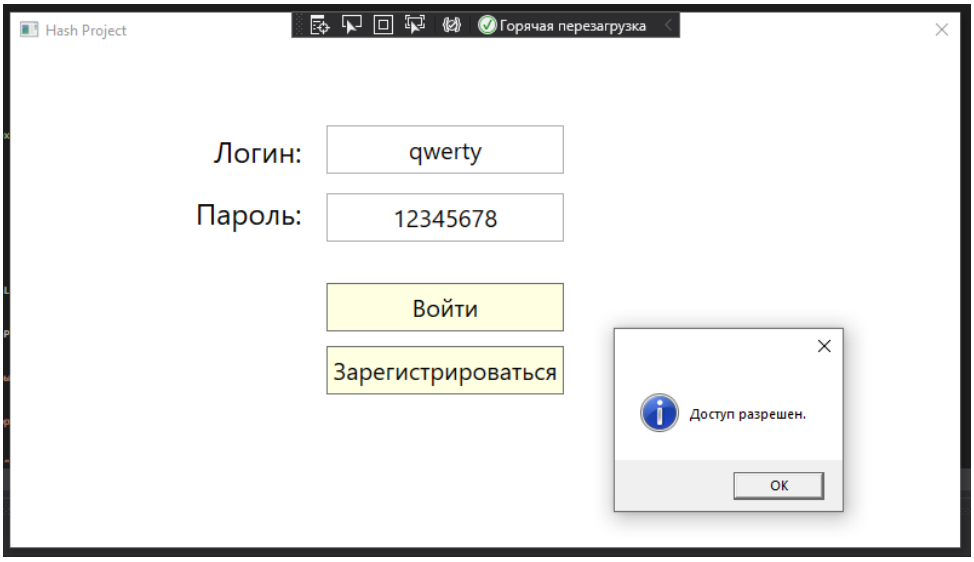
Работа приложения

Здесь включены скриншоты, отражающие работу программы.









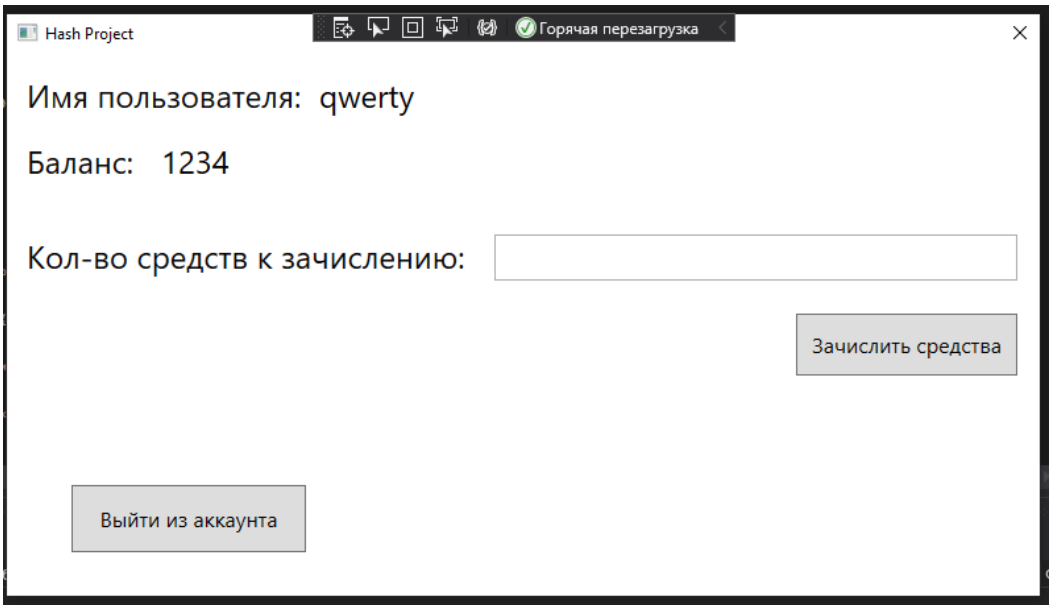


рис.5.3–5.7 Работа приложения