*Содержание*

*Введение*2

3

5

6

6

7

11

14

14

17

20

25

25

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](https://github.com/AlexBimlnk/HashProject).

*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.

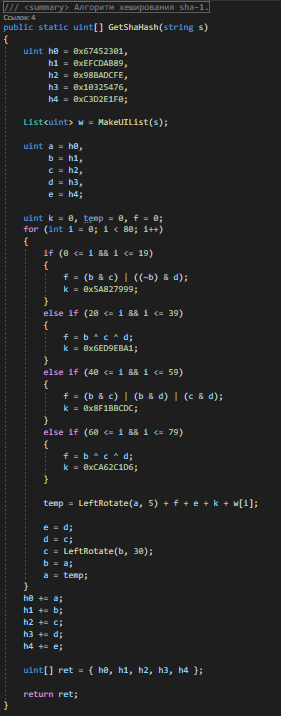


рис.2.1 Алгоритм SHA-1

**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.

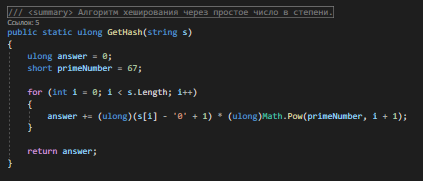


рис.2.2 Второй алгоритм хеширования

**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.

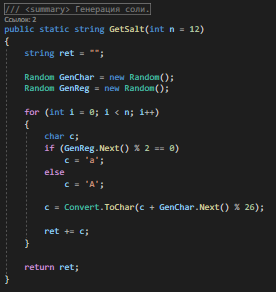


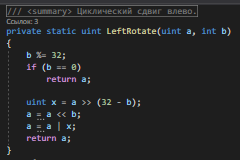
рис.2.3 Получение соли

*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.



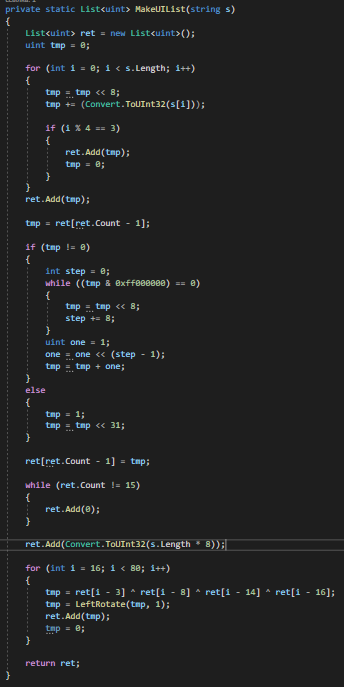


рис.2.4 - 2.5 Вспомогательные методы

**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)*.

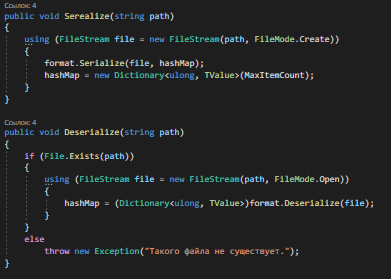


рис.2.6 Сериализация и десериализация

**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.

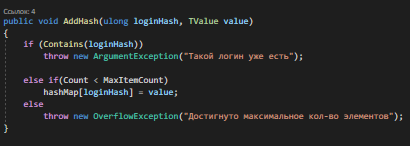
****

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*.

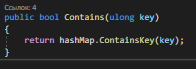


рис.2.8 Метод *Contains(ulong key)*

**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*.

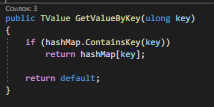


рис.2.8 Метод *GetValueByKey(ulong key)*

*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.

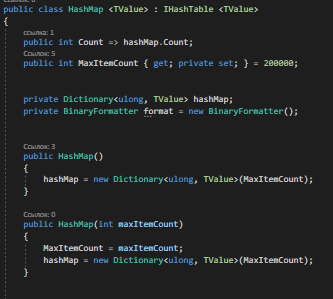


рис.2.9 Свойства *HashMap*

*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.

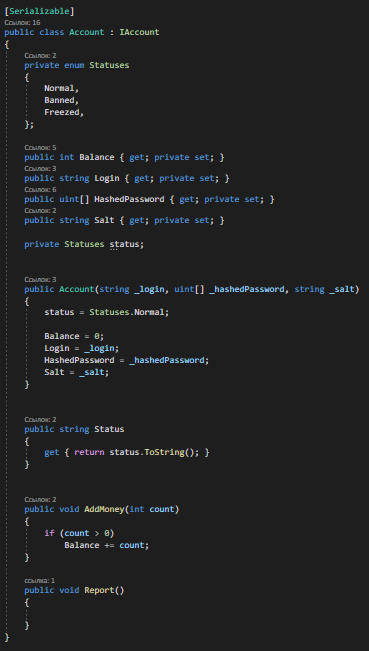


рис.3Класс *Account*

**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.

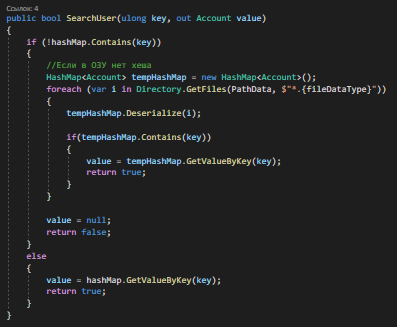


рис.3.1 Поиск пользователей в локальной «базе данных»

**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.

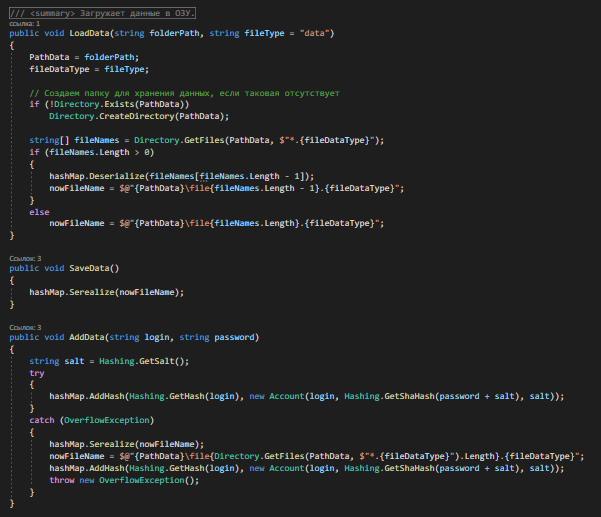


рис.3.2Методы *LocalDataMager’a*

*Свойства*

**PathData**

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

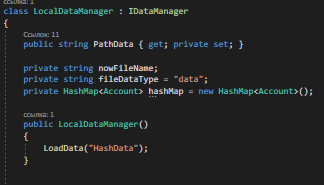


рис.3.3Свойство *PathData*

**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.

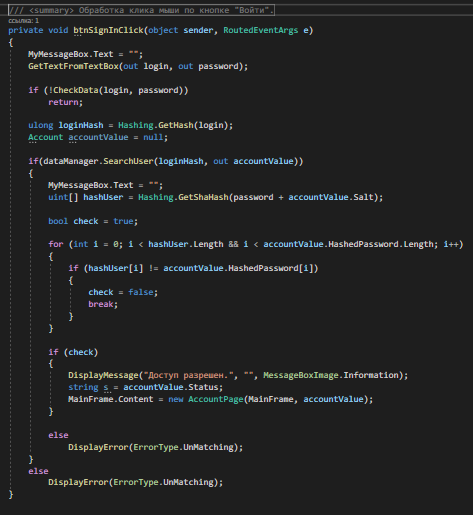


рис.4Кнопка «Войти»

**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».

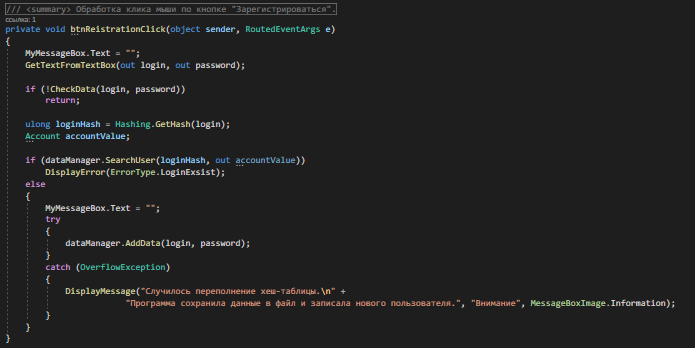


рис.4.1Кнопка «Зарегистрироваться»

**Кнопка «Зачислить средства» (Располагается на форме аккаунта)**

Метод addMoneyBtn\_Click

Проверяет вводимые данные и зачисляет деньги на баланс.

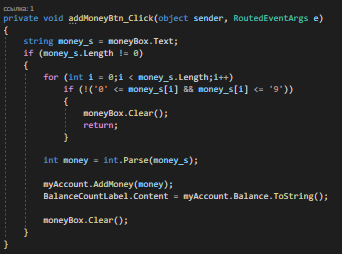


рис.4.2Кнопка «Зачислить средства»

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

Метод exitBtn\_Click

Возвращает пользователя на главную форму.

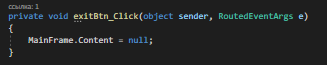
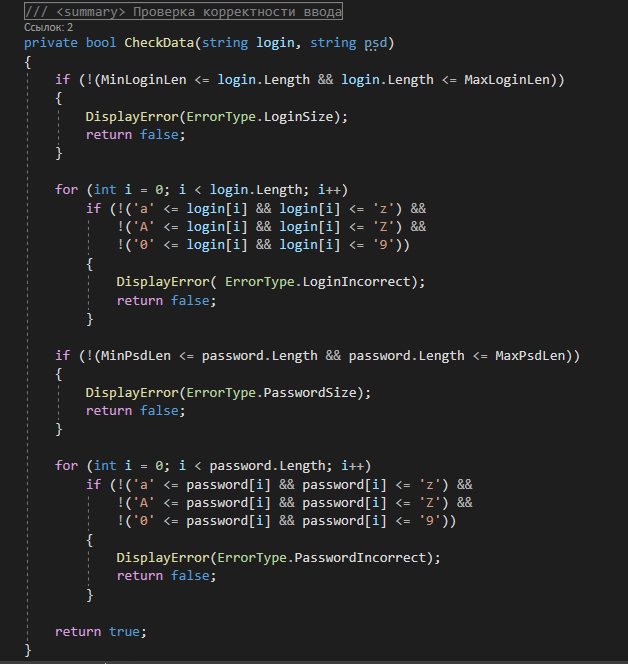


рис.4.3 Кнопка «Выйти из аккаунта»

*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.



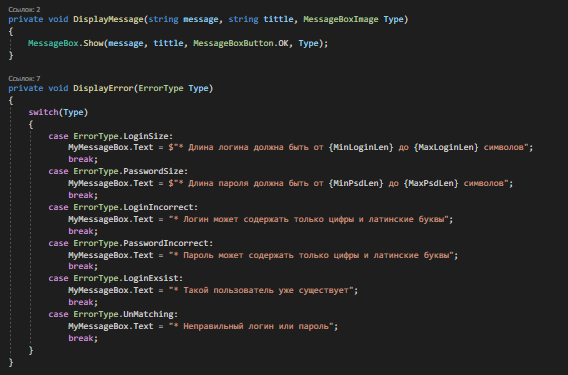


рис 4.4–4.5 Методы для проверки входных данных и обработки ошибок.

*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*



рис.5 Логика тест метода

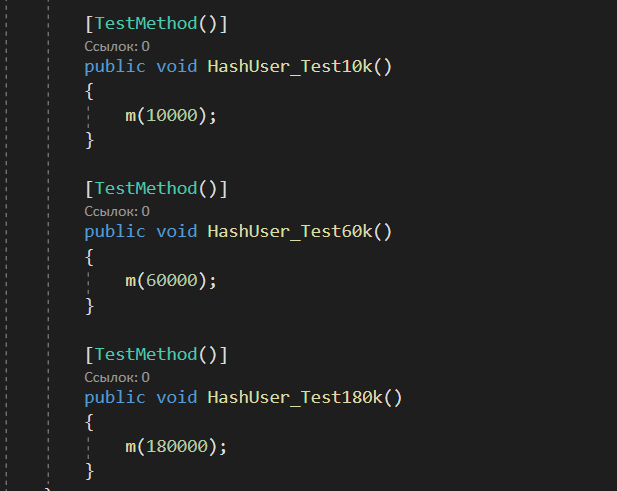
*Тест методы*

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.

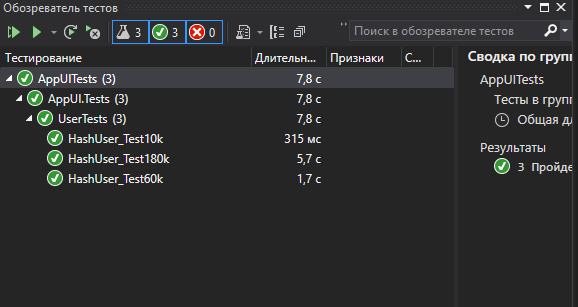
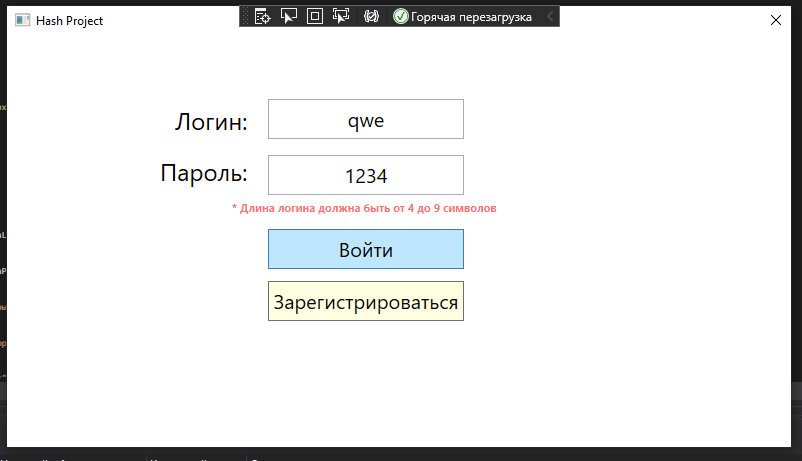
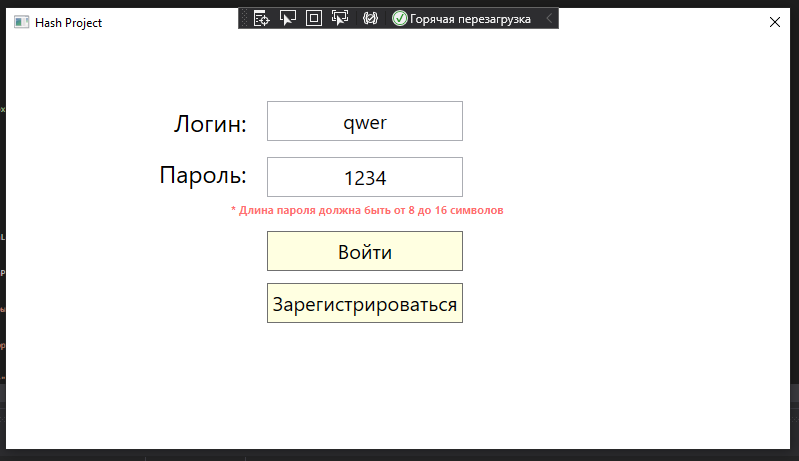


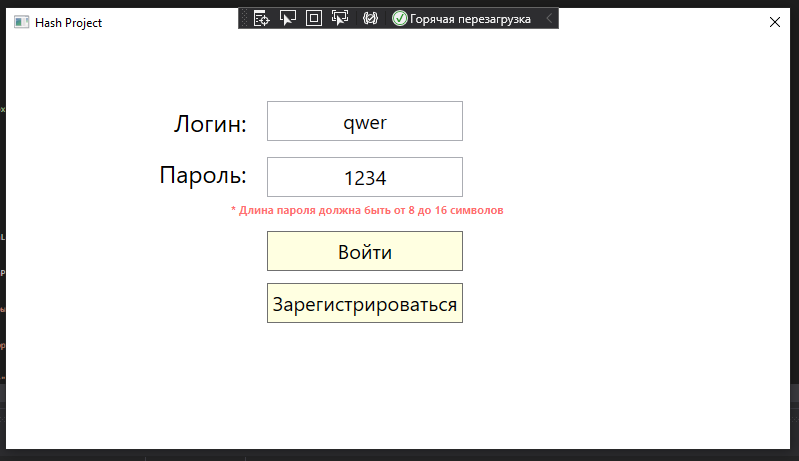
рис.5.2 Результаты тестирования

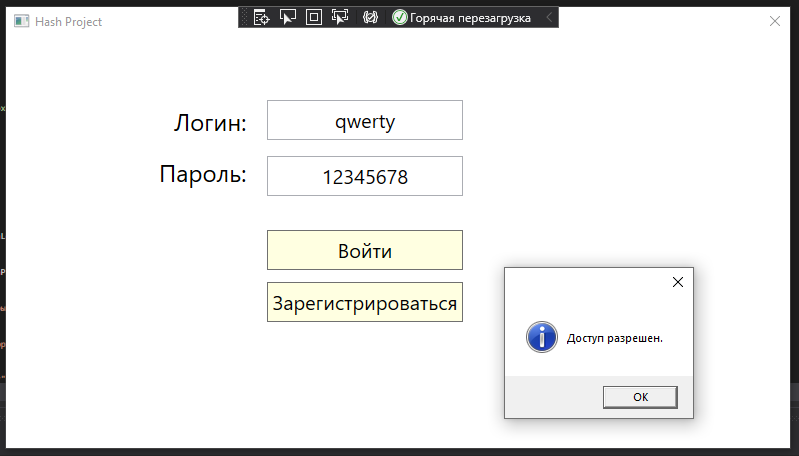
*Работа приложения*

Здесь включены скриншоты, отражающие работу программы.









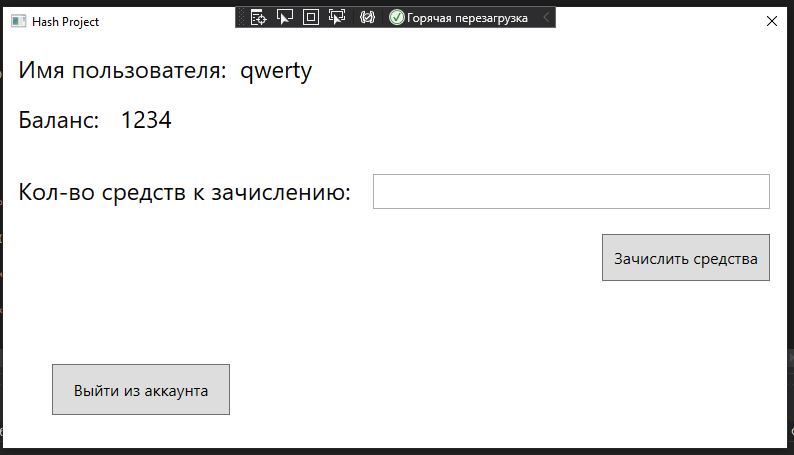


рис.5.3–5.7 Работа приложения