

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

**«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ»**

В. В. Воронина, В.С. Мошкин

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ АНАЛИЗА СЛАБОСТРУКТУРИРОВАННЫХ ИНФОРМАЦИОННЫХ РЕСУРСОВ

Учебное пособие

Ульяновск 2015

УДК 681.3
ББК 32.973-018.1
Р 12

Рецензенты:
профессор кафедры «Информационные технологии» УлГУ,
д.т.н. И. В. Семушин;
доцент кафедры «Вычислительная техника» УлГТУ, к.т.н.
К. В. Святков.

Утверждено редакционно-издательским советом
университета в качестве учебного пособия

Разработка приложений для анализа
Р 12 слабоструктурированных информационных ресурсов :
учебное пособие/ В. В. Воронина, В.С. Мошкин – Ульяновск :
УлГТУ, 2015. – 150 с.

Учебное пособие рассматривает статистические, кластерные и онтологические алгоритмы анализа текстов, а также описывает способы их реализации в приложениях на языке C#.

Пособие предназначено для студентов направления 09.04.04 «Программная инженерия», а также для студентов других направлений, изучающих дисциплины, связанные с анализом текстовой информации.

УДК 681.3
ББК32.973.2-018.2

© Колл. авторов, 2015.
© Оформление. УлГТУ, 2015.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
БАЗОВЫЙ СИНТАКСИС ЯЗЫКА C#	8
Комментарии.....	8
Объявление переменных	8
Классы и объекты.....	8
Объявление класса	9
Создание объектов	10
Процедурные и функциональные методы в C#	11
Некоторые базовые операции	13
Массивы	14
Цикл while	16
Цикл do ... while	16
Цикл for	17
Цикл foreach	17
Условный оператор if.....	18
Параметрические классы.....	18
Класс Math.....	19
Строки и символы в C#.....	20
Класс Char	21
Класс String	23
Класс StringBuilder	28
Контрольные вопросы к разделу	30
Практические задания к разделу	30
СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ	31
Статистический метод Mutual Information	35
Статистический метод T-Score	38
Разработка приложения на языке C#.....	40
Построение графиков средствами C#	50
Работа с файлами.....	54
Контрольные вопросы к разделу	56

Практические задания к разделу	57
МОРФОЛОГИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ	58
Контрольные вопросы к разделу	69
Практические задания к разделу	69
КЛАСТЕРНЫЙ АНАЛИЗ ТЕКСТОВ	70
Контрольные вопросы к разделу	85
Практические задания к разделу	86
ОНТОЛОГИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ.....	87
Контрольные вопросы к разделу	111
Практические задания к разделу	111
РЕАЛИЗАЦИЯ ДОПОЛНИТЕЛЬНЫХ ВОЗМОЖНОСТЕЙ.....	112
Вывод отчетов в формате PDF.....	112
Работа с текстами для анализа в формате Word	120
Работа с текстами для анализа в формате OpenOffice	125
Работа с электронной почтой.....	132
Контрольные вопросы к разделу	136
Практические задания к разделу	136
ЗАКЛЮЧЕНИЕ	137
Приложение	138
ГЛОССАРИЙ	144
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	146
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	147

ВВЕДЕНИЕ

В последние десятилетия наблюдается неуклонный рост текстовой информации, нуждающейся в обработке. Причем, быстрой, эффективной и автоматизированной. Помимо обычного поиска по ключевым словам все чаще стали возникать задачи выделения данных из информационного ресурса, или его семантики, а также осуществление на основе нее каких-либо манипуляций.

В библиотечном деле, лексикографии и терминоведении очень часто возникает задача извлечения терминологии из текста. Прикладные аспекты ее автоматизированного решения имеются в информационном поиске и машинном обучении. При этом под «извлечением терминологии» мы понимаем обработку текста и формирование списка терминов-кандидатов для добавления в словарную базу. В результате проведения этой операции мы избавляемся от терминологической избыточности и добиваемся последовательности в терминологии. Что особенно актуально в переводческих задачах и задачах информационного обеспечения систем автоматизированного проектирования (САПР) при необходимости анализа больших массивов документации и унификации используемой терминологии.

Принцип работы существующих алгоритмов извлечения терминов основан на статистических и лингвистических методах. В основе первых – лежит вычисление степени терминологичности на основании числовых закономерностей, присущих термину или нетермину. В основе вторых – лежит отбор по определенным лексико-грамматическим шаблонам и другим лингвистическим признакам термина [1].

При работе с большими предметными областями и при обработке больших корпусов текстов даже лучшие методы извлечения терминологических словосочетаний показывают значительное падение процентного содержания терминов с 90% на первой сотне списка извлеченных терминологических словосочетаний до 60% на третьей

тысяче [2]. Поэтому в настоящее время актуальной является задача разработки семантических алгоритмов для улучшения качества упорядочения списков словосочетаний с целью повышения процентной доли терминов в начале списка. Однако большинство теоретических разработок в этой области изложено в рамках научных статей, в которых практической части уделено крайне мало внимания. В связи с этим возникает задача доступного и понятного описания реализации алгоритмов анализа текстов.

Представленный в учебном пособии материал будет полезен студентам-программистам, так как описывает теоретические аспекты задачи, а также студентам-лингвистам, так как стиль изложения ориентирован на базовый уровень программирования и отсутствие знакомства со средой Visual Studio и редактором Protege. Для простоты освоения этих инструментов в тексте приведены скриншоты, позволяющие студентам быстро сориентироваться в незнакомом интерфейсе.

В данном учебном пособии рассмотрены статистические, а также интеллектуальные методы извлечения терминологии, и описана их реализация на языке C#. Кроме того приведен обзор полезных возможностей и технологий, которые могут быть использованы для разработки приложений анализа слабоструктурированных информационных ресурсов.

В первом разделе представлен базовый синтаксис используемого языка программирования. Второй раздел посвящен описанию разработки приложения для статистического анализа текстов. В третьем разделе описана работа с программой Mystem – инструментом морфологического анализа текстов. Четвертый и пятый разделы посвящены кластерному и онтологическому анализу соответственно. В шестом разделе описана реализация дополнительных возможностей, таких как работа с различными форматами хранения текстовой информации, а также работа с электронной почтой и создание отчетов

по результатам анализа. После каждого раздела представлен список контрольных вопросов и практические задания.

С представленным материалом рекомендуется работать следующим образом. Информацию первого раздела следует рассматривать как справочную и необходимую для понимания листингов кода в книге. Для тех, кто владеет языком C# даже на базовом уровне, этот раздел не обязателен к прочтению. Им рекомендуется выполнить представленные практические задания для понимания, с каким видом информации и как придется в дальнейшем работать.

Начиная со второго раздела студентам рекомендуется сначала воспроизвести описанный процесс разработки тренировочного приложения, а затем оптимизировать предложенный код и выполнить соответствующие практические задания. Ориентируясь на приведенные скриншоты можно проверять правильность работы воспроизведенного кода.

Учебное пособие предназначено для студентов направления 09.04.04 «Программная инженерия», изучающих дисциплину «Интеллектуальные САПР» и «Теоретические основы программной инженерии», а также для студентов других направлений, решающих задачи разработки приложений для анализа текстов.

Книга подготовлена по результатам исследования в рамках государственного задания №2014/232 на выполнение государственных работ в сфере научной деятельности Минобрнауки России по проекту «Разработка нового подхода к интеллектуальному анализу слабоструктурированных информационных ресурсов».

БАЗОВЫЙ СИНТАКСИС ЯЗЫКА C#

Данное учебное пособие ориентировано на описание реализации различных алгоритмов анализа текстовой информации в среде Visual Studio. Поэтому вашему вниманию представляется описание основ этого языка C#, выбранного из множества языков среды.

Комментарии

```
// Для того чтобы закомментировать строку кода,  
// перед ней ставим два слеша.  
/* А еще можно просто написать текст между слешом и звездочкой.*/
```

Объявление переменных

Для выполнения лабораторных работ вам могут потребоваться следующие виды переменных:

```
int a=0; //целочисленная переменная a  
string b=" "; //строковая переменная b  
char c=' '; //символьная переменная c  
double r=0; //дробная переменная  
bool y=true; //логическая переменная y  
string[] ms; // массив строк
```

Классы и объекты

Прежде чем перейти к дальнейшему описанию языка, рассмотрим определения нескольких ключевых понятий. С теоретической точки зрения:

Класс – это тип, описывающий устройство объектов.

Поля – это переменные, принадлежащие классу.

Методы – это функции (процедуры), принадлежащие классу.

Объект – это экземпляр класса, сущность в адресном пространстве компьютера.

Можно сказать, что класс является шаблоном для объекта, описывающим его структуру и поведение. Поля класса определяют структуру объекта, методы класса – поведение объекта.

С точки зрения практической реализации (в самом тексте программы) класс является типом данных, а объект – переменной этого типа.

Объявление класса

Объявление класса состоит из двух частей: объявление заголовка класса и объявление тела класса. Заголовок класса состоит из модификатора доступа, ключевого слова `class` и имени самого класса. Тело класса – есть конструкция, заключенная в фигурные скобки и содержащая объявление полей и методов, принадлежащих классу.

Пример объявления класса:

```
public class MyClass { int a; }
```

Так как в одной программе может быть множество классов, некоторые из которых могут выполнять схожие функции, то для удобства навигации было придумано понятие пространства имен.

Пространство имен (или `namespace`) – средство логической группировки классов программы. Допустим, мы пишем программу для автоматизации деятельности какого-либо магазина. Нам необходимо будет вести учет товара, фиксировать оплаты от клиентов и работать с поставщиками. Логично будет классы, работающие с товарами объединить в одно пространство имен, с покупателями в другое, с поставщиками в третье. Тогда, даже если функции какого-либо типа разобьются по нескольким файлам, мы будем знать, к какому типу они относятся. Или наоборот, если внутри одного файла будут функции нескольких типов, то по пространствам имен мы их разделим.

Пространство имен объявляется следующим образом:

```
namespace <Имя пространства имен>{ <Объявления классов> }
```

Внутри пространства имен к объявленному в нем классу мы можем обращаться просто по имени. Если же мы вызываем класс, определенный в другом пространстве имен, то обращаться к нему мы должны так: `<Имя пространства имен>.<Имя класса>`. Средством, которое позволяет сокращать имена классов, является оператор

```
using <ИмяПространстваИмен>;
```

Он используется в самом начале текста программы и, по сути, является подключением обозначенных пространств имен к проекту. В C# основные классы находятся в пространстве имен System, поэтому в каждом проекте неизбежно встретится строка

```
using System;
```

Объявление объекта (создание объекта как экземпляра класса) состоит из двух частей: создание переменной-ссылки на область памяти, в которой будет располагаться объект, выделение памяти для объекта и заполнение этой памяти начальными значениями, иначе говоря, инициализация данной переменной-ссылки. Объявление переменной-ссылки, а иными словами объекта, подчиняется общему правилу объявления переменных в C#. Напомним, что переменные могут объявляться в любом месте в теле методов. Переменная, объявленная вне тела метода, но внутри тела класса, становится полем.

Пример объявления переменной-объекта класса MyClass:

```
MyClass MyObj;
```

При объявлении переменная может быть сразу инициализирована (ей может быть присвоено какое-либо значение). Пример инициализации переменной при объявлении:

```
int a=0;
```

Для переменной-объекта ее инициализация будет называться созданием объекта.

Создание объектов

Под созданием объекта будем понимать выделение под него памяти и заполнение ее значениями объявленных в классе полей.

Выделение памяти осуществляет оператор new, а задачу заполнения памяти начальными значениями решает специальный метод объекта, называемый конструктором. Конструктор – метода объекта, объявленный следующим образом: для этого метода всегда используется модификатор доступа public, нет типа возвращаемого

значения (нет даже `void`), имя метода совпадает с именем класса. Однако компилятор `C#` не требует обязательного определения конструктора для класса. Если конструктор не объявлен, компилятор вызовет так называемый конструктор по умолчанию, который создаст сам.

Таким образом, создание объекта класса `MyClass` будет иметь вид:

```
MyClass MyObj=new MyClass();
```

Процедурные и функциональные методы в `C#`

Функциональные методы – методы, которые в результате своей работы возвращают какое-либо значение.

Процедурные методы – методы, которые в результате своей работы не возвращают никакого значения.

Синтаксис описания функционального метода:

```
<модификатор доступа> <тип возвращаемого значения> <имя метода>
(<список параметров>)
{
    <....тело метода ....>
    return <значение>;
}
```

Синтаксис описания процедурного метода:

```
<модификатор доступа> void <имя метода> (<список параметров >)
{
    <....тело метода ....>
}
```

`void` – специальное ключевое слово, обозначающее пустой тип. Список параметров, если он не пустой, состоит из перечисленных через запятую пар `<тип имя>`. Ключевое слово `return` может использоваться и в процедурном методе, если необходимо по какому-то условию экстренно прервать его работу, но таким образом используется редко. Необходимо отметить, что код метода, расположенный после этого ключевого слова никогда не будет выполнен.

Пример объявления класса с процедурным и функциональным методом приведен ниже:

```

class MyClass {
public void ProcMet(int a,int b){ //тело метода...}
public int FuncMet() { //тело метода...
return числовое_значение; }
}

```

Вызов методов класса осуществляется через имя объекта, или через имя класса, если метод был объявлен со специальным ключевым словом `static`. Для класса, приведенного выше, вызов будет иметь вид:

```

MyClass MyObj=new MyClass();
int a=MyObj.FuncMet();
MyObj.ProcMet(4,5);

```

Если же метод объявлен следующим образом:

```

class MyClass {
public static void ProcMet(int a,int b){ //тело метода...}
}

```

то его вызов будет осуществляться так:

```

MyClass.ProcMet(4,5);

```

Единственный метод класса, вызов которого происходит по-другому – это конструктор. Управление ему передается автоматически, используя ключевое слово `new`. Никаким другим образом конструктор вызывать нельзя.

При разработке методов могут возникать неоднозначности обращения к элементам, если поле класса и параметр в методе имеют одно и то же имя. Причем, в этом случае параметр перекроет видимость поля. Например, пусть класс объявлен следующим образом:

```

class Class1 {
    int a;
    public void Method(int a){    a = a;    }
}

```

Ошибки компиляции в данном случае не будет. Только предупреждение с текстом: «Assignment made to same variable; did you mean to assign something else? ». Компилятор укажет на опасный момент, но предоставит принимать решение программисту: вдруг он на

самом деле имел в виду то, что написал и решил присвоить значение параметра ему же. Для разрешения подобных конфликтов используется ключевое слово `this`. То есть если мы хотим полю «а» присвоить значение параметра «а», то код будет следующим:

```
class Class1
{
    int a;
    public void Method(int a)
    {
        this.a = a;
    }
}
```

В теле метода ключевое слово `this` хранит указатель на объект, вызвавший этот метод.

Некоторые базовые операции

В таблице 1 приведен список основных арифметических операций с примерами использования.

Таблица 1. Арифметические операции

Значок операции	Назначение	Пример использования
+	Сложение	Запишем в переменную <code>c</code> результат сложения значений <code>a</code> и <code>y</code> : <code>int a=10; int y=12; int c=a+y;</code>
-	Вычитание	Запишем в переменную <code>c</code> результат вычитания <code>y</code> из <code>a</code> : <code>int a=10; int y=12; int c=a-y;</code>
++	Увеличение на 1	Увеличим переменную <code>c</code> на 1: <code>int c=10; c++;</code>
--	Уменьшение на 1	Уменьшим переменную <code>c</code> на 1: <code>int c=10; c--;</code>
%	Остаток от деления	Проверим, делится ли переменная <code>a</code> на переменную <code>c</code> без остатка: <code>int a=10; int c=2; if (a%c==0) //Да; else //Нет;</code>

Окончание таблицы 1

Значок операции	Назначение	Пример использования
==	Проверка равенства на	Проверим, равны ли а и с: <pre>int a=10; int c=2; if (a==c) //Да; else //Нет;</pre>
!=	Проверка различия (не равно) на	Проверим, различны ли а и с: <pre>int a=10; int c=2; if (a!=c) //Да; else //Нет;</pre>

Кроме перечисленных выше операций, вам, возможно, понадобятся такие как: *(умножение),/(деление),<(меньше),>(больше). Они работают аналогично.

Кроме базовых операций язык C# предоставляет вам возможность использовать операции-сокращения, состоящие из знака операции и знака равно. Например, рассмотрим использование сокращенной операции «+=» на следующем примере:

```
int a=5;
a=a+5; //обычная операция +
a+=5;  //аналогичная примененной выше сокращенная операция +=
```

Массивы

Массив – упорядоченное множество однотипных элементов. В C# определены три различных категории массивов: одномерные, прямоугольные и вложенные. Второй вид представляет собой массив, элементами которого также являются массивы.

В C# массив является ссылочным типом, то есть в программе переменная-массив – ссылка на область памяти, где этот массив фактически хранится. Поэтому при работе с массивами обязательно

использовать оператор new для выделения памяти при создании нового массива.

Примеры ссылок на массив:

```
string[] arr; // одномерный массив  
string[,] arr1; // прямоугольный двумерный массив  
string[][] arr2; // одномерный массив одномерных массивов
```

Примеры создания одномерного, прямоугольного и вложенного массивов:

```
string[] arr=new string[7];  
string[,] arr1 = new string[2,2];  
string[][] arr2= new string[2][];  
arr2[0]=new int[2];  
arr2[1]=new int[5];
```

На рисунке 1 наглядно представлены все объявленные массивы. Обратите внимание, что во вложенном массиве размеры элементов-массивов могут быть разными, но под каждый из них обязательно выделять память.

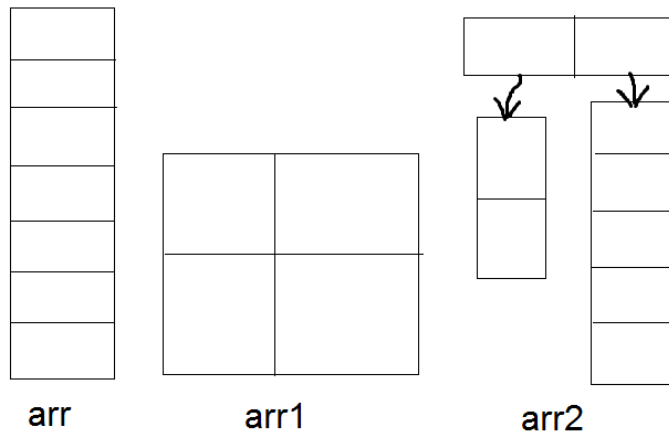


Рис. 1. Графическое представление массивов

При создании массива его можно сразу инициализировать, например, вызвав некий функциональный метод, возвращающий массив:

```
string[] arr = MyObj.ReturnArray(8);
```

Для обращения к элементам массива мы пишем его имя, а затем в квадратных скобках указываем нужный индекс. Приведем пример кода записи числа в первые элементы всех массивов:

```
arr[0]=«строка»; arr1[0,0]=«строка»; arr2[0][0]=«строка»;
```

Для массивов ключевым понятием является его длина. В одномерном массиве значение длины можно получить конструкцией:

```
имя_массива.Length
```

В многомерном массиве предыдущий вариант вернет количество всех элементов массива (то есть для массива 2 на 2 значение будет 4). Для того чтобы получить размер каждого отдельного измерения нужно использовать конструкцию:

```
имя_массива.GetLength(номер_измерения) ;
```

Цикл while

Это цикл с предусловием. Имеет следующий синтаксис:

```
while (УсловиеПродолжения) {Операторы; }
```

Пример

```
int i=0;  
while(i<=10) i++;
```

Работает по следующему правилу: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Цикл do ... while

Цикл с постусловием. Синтаксис:

```
do {Операторы;} while (УсловиеПродолжения)
```

Пример

```
int i=0;  
do i++; while(i<=10) ;
```

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения.

Цикл for

Пошаговый цикл. Записывается как:

```
for ([Выражение_Инициализации] ; [Условие_Продолжения] ; [Выражение_Шага])
```

Выражение_Инициализации, Условие_Продолжения, Выражение_Шага в могут быть пустыми, но наличие пары символов ';' в заголовке цикла обязательно.

Пример синтаксиса:

```
for(int i=??;i<??;i++){//что-то тут делается; }
```

В данном случае `int i=??` – установка начального значения переменной, которая будет меняться в цикле, `i<??` или `i>??` – установка условия завершения цикла, `i++` или `i--` – шаг изменения переменной.

Пример самого цикла:

```
for(int i=0;i<10;i++) //код вывода i;
```

Цикл foreach

Для перебора элементов массива можно использовать различные виды циклов. Если же вам необходимо только считать элементы (не изменить их, а только просмотреть), то можно использовать универсальный перечислитель для коллекций – цикл `foreach`.

Синтаксис:

```
foreach(<переменная_элемент_коллекции> in <коллекция>)
```

Цикл имеет следующую семантику «Для каждого элемента из коллекции делать». Так как массив можно определить как коллекцию, то этот цикл может использоваться для перебора элементов массива.

Пример:

```
string[] ar=new string[9];  
foreach(string i in ar) //код вывода i;
```

Основная особенность этого цикла заключается в том, что внутри его тела нельзя изменять элементы коллекции. Они доступны только для чтения.

Условный оператор if

Условный оператор if имеет следующие правила использования. После ключевого слова if располагается взятое в круглые скобки условное выражение (булево выражение), следом за которым располагается оператор (блок операторов) произвольной сложности. Далее в операторе if ... else ... после ключевого слова else размещается еще один оператор (блок операторов).

Пример

```
if (i>0)
    //действие, если условие истинно
else
    //действие, если условие ложно
```

Параметрические классы

В данном пособии вам встретится конструкция параметрического класса. Хотя она и упомянута вскользь, рассмотрим, что она собой представляет. Разработаем параметрический класс, имеющий поле произвольного типа:

```
class Example<T> { public T a; }
```

В данном случае T – параметр, обозначающий тип данных. Если нужно использовать несколько типов, то параметры перечисляются через запятую. Рассмотрим теперь создание объекта данного класса с числовым полем:

```
Example <int> s=new Example <int>();
s.a=6;
```

и строковым:

```
Example <string> s=new Example <string>();
s.a="6";
```

То есть классы этого вида позволяют работать с любыми типами данных. В данном пособии вам встретится вспомогательный параметрический класс List, представляющий собой динамический массив объектов произвольного типа.

Коллекция List по сути представляет собой динамический массив, основное достоинство которого заключается в том, что не нужно сразу указывать размер. Объекты в списке могут быть любого типа.

Синтаксис объявления List:

```
List<Тип_значения> имя;
```

Конструктор объекта (кроме конструктора по умолчанию) может принимать либо количество элементов в списке, либо ссылку на коллекцию, элементы которой нужно поместить в список.

Добавление элемента в List осуществляется методом Add(значение). Количество элементов коллекции возвращается свойством Count. Удаление всех элементов из коллекции осуществляется методом Clear(). Поиск элемента в коллекции осуществляется методом Contains(значение), но этот метод вернет только ответ, есть или нет элемент в коллекции. Удаление элемента осуществляется методами Remove (значение), RemoveAt (индекс), RemoveAll (метод_сравнения), RemoveRange (начальный_индекс, количество_значений).

Для типов-ссылок метод Remove производит удаление по ссылке, поэтому если из списка нужно удалить объекты с одинаковыми полями, то нужно использовать метод RemoveAll, имея в виду, что он удалит все вхождения подходящих объектов. Рассмотрим пример удаления всех положительных элементов из списка. Пусть в классе объявлен метод, внутри которого создается коллекция чисел List, заполняется десятью случайными элементами и затем из нее удаляются все положительные элементы. Обратите внимание, что в классе нам дополнительно приходится объявить метод, определяющий положительное число или нет и ссылку на него передавать в метод RemoveAll.

Класс Math

Для подсчета статистических коэффициентов вам могут потребоваться такие математические функции как логарифм, степень, корень и т.д. Все это можно найти в классе Math. В нем все методы

определены как статические, а также сам класс закрыт от наследования и создание объекта не поддерживает. Он содержит два свойства – значения числа Π и E . Все остальное реализовано с помощью методов. В таблице 2 приведены основные методы. Тип принимаемых и возвращаемых значений в большинстве методов – `double`.

Таблица 2. Методы класса Math

Название	Описание
Sin, Cos, Tan	Тригонометрические методы вычисления синуса, косинуса и тангенса.
ASin, ACos, ATan, ATan2	Методы, вычисляющие обратные тригонометрические функции.
Tanh, Sinh, Cosh	Методы, вычисляющие гиперболические функции.
Exp, Log, Log10	Методы, вычисляющие экспоненту и логарифмические функции.
Abs, Sqrt, Sign	Методы, вычисляющие модуль, корень и знак.
Ceiling, Floor, Round;	Методы, выполняющие округление.
Min, Max, Pow, IEEERemainder.	Методы, вычисляющие минимум, максимум, степень и остаток от деления.

Строки и символы в C#

Так как данное пособие посвящено описанию разработки приложений для анализа текстов, нам необходимо рассмотреть какие средства предоставляет C# для работы с текстовой информацией.

В данном языке программирования представлено три вида строк. Это одиночные символы, строки постоянной длины и строки переменной длины. Причем, для работы со строками переменной длины

помимо типа представляются еще два класса: String и StringBuilder. Для работы с отдельными символами также есть отдельный класс – Char.

Одиночный символ представляется типом char и является частным случаем строки – строкой длиной в 1 символ. Он полезен во многих задачах, так как фактически является составляющим любой строки. Чаще всего над строками выполняются операции разбора и сборки, поэтому обычно приходится работать с отдельными символами.

При работе с символами необходимо всегда помнить, что преобразования между типами int и char происходят неявно. Например, код

```
char ch='9';  
int a = 3 + ch;
```

откомпилируется и выполнится без ошибок, однако, в переменной a будет сохранено не число 12, а код символа 9 плюс 3.

В самом простом случае строка может быть представлена как массив символов:

```
char[] stroka;
```

Тогда любая операция со строкой эквивалентна операцией с массивом, но такой подход хорош только для строк постоянной длины. Так как обычный массив не приспособлен для вставки или удаления элементов.

Чаще всего при работе со строками используется тип string, определяющий строки переменной длины. С переменными этого типа можно выполнять разнообразные действия: поиск вхождения одной строки в другую, вставку, замену и удаление подстрок.

Для работы с каждым из видов строк описаны специальные типы данных, которые обладают особыми характеристиками и функциональностью.

Класс Char

Это класс, использующий двухбайтную кодировку Unicode для представления символов. Для него в языке C# определены символьные

константы, которые можно задать не только символом или escape-последовательностью, заключенными в одинарные кавычки, но и Unicode-последовательностью, задающей код символа.

Приведем примеры объявления и инициализации символьных переменных:

```
char ch = new Char();  
char ch1='a';  
char ch2 ='\x4A';  
char ch3='\u0057';
```

Для работы с символами класс Char содержит различные статические методы. Рассмотрим их описание более подробно.

Метод GetUnicodeCategory возвращает Unicode категорию символа. В таблице 3 приведены эти методы и названия соответствующих категорий.

Таблица 3.Unicode-категории

Имя метода	Категория
IsControl	Управляющие символы
IsDigit	Десятичные цифры
IsLetter	Буквы
IsLetterOrDigit	Буквы или цифры
IsLower	Символы в нижнем регистре
IsNumber	Десятичные или шестнадцатеричные цифры
IsPunctuation	Знаки препинания
IsSeparator	Разделители
IsSurrogate	«Суррогатные» символы
IsUpper	Символы в верхнем регистре
IsWhiteSpace	«Белые пробелы»

Более полное представление об этих категориях можно составить, рассмотрев методы вида Is<название категории>, которые возвращают логическое значение, говорящее о принадлежности символа к указанной категории.

Метод `GetNumericValue` предназначен для преобразования значения символа в соответствующую цифру. Если преобразование невозможно, метод вернет значение -1.

Метод `Parse` преобразует строку в символ. Если преобразование невозможно, генерируется исключение.

Методы `ToLower` и `ToUpper` приводят символ к нижнему и верхнему регистру соответственно.

Как было сказано выше, на основе класса `Char` можно создавать строки постоянной длины, работая с ними как с массивами и объявляя соответствующим образом. В данном случае мы получим экземпляр класса-наследника базового класса `Array` со всеми доступными ему методами. Подробнее об этом классе мы поговорим ниже.

Класс `String`

Объекты данного класса представляют строки. Создавать их можно как с вызовом конструктора, так и без. Класс `String` содержит три перегруженных конструктора. Рассмотрим пример создания строк всеми четырьмя способами:

```
string s = "world"; //без использования конструктора
string s1 = new string('s', 5); //символ, повторяющийся 5 раз
char[] chm = {'q','w','e','r','t','y'};
string s2 = new string(chm); //на основе массива символов
string s3 = new string(chm, 0, 3); //на основе части массива
```

строками определены операции присваивания (`=`), проверки эквивалентности (`==` и `!=`), взятие индекса (`[]`), а также конкатенация или сцепление строк (`+`).

При выполнении операции присваивания необходимо помнить, что тип `string` – тип-ссылка. Причем, ссылка на объект неизменяемого класса. Неизменяемым называется класс, для которого невозможно изменить значение объекта при вызове его методов. При вызове динамических методов может создаваться новый объект, но изменение значения существующего объекта невозможно. Операция присваивания будет работать следующим образом: при ее выполнении создается

ссылка на константную строку, расположенную в куче. С этой константой может быть связано несколько переменных строкового типа, но так как тип `string` – неизменяемый, то когда одна из переменных получает новое значение, она связывается с новой константой в куче, а остальные так и ссылаются на старый объект.

Хотя строка и является ссылочным типом, но операции, проверяющие эквивалентность, сравнивают значения строк, а не ссылок.

Операция взятия индекса позволяет рассматривать строку как массив символов, элементы которого доступны для чтения, но недоступны для записи.

Операция конкатенации в ходе своей работы сцепляет две строки, приписывая вторую строку к хвосту первой, и создает новую константную строку в куче.

Как и любого класса, у `String` определено множество статических и динамических методов. Ниже в таблице 4 приведено описание наиболее важных методов.

Таблица 4. Методы класса `String`

Имя метода	Параметры	Описание
Динамические методы		
Insert	Стартовый индекс(int) Вставляемая подстрока(string)	Метод вставляет подстроку в исходную, начиная с заданной позиции, и возвращает ссылку на новую строку.
Replace	Старое значение(char или string) Новое значение(char или string)	Метод заменяет все вхождения указанной подстроки новой подстрокой и возвращает ссылку на новую строку.

Имя метода	Параметры	Описание
Динамические методы		
Remove	Стартовый индекс(int) Количество удаляемых символов(int)	Метод удаляет заданное количество символов из исходной строки, начиная с заданной позиции, и возвращает ссылку на новую строку. Если требуемое количество символов не указано, удаление символов происходит до конца строки.
Substring	Стартовый индекс(int) Количество копируемых символов(int)	Метод копирует заданное количество символов из исходной строки, начиная с заданной позиции, и возвращает ссылку на новую строку. Если требуемое количество символов не указано, копирование символов происходит до конца строки.

Имя метода	Параметры	Описание
Динамические методы		
IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny	Искомое значение(char, char[] или string) Стартовый индекс(int) Количество просматриваемых символов(int) Правила сравнения(StringComparison)	Методы определяют индексы первого или последнего вхождения заданной подстроки или любого символа из заданного набора. Возвращают -1, если искомое не найдено.
StartsWith, EndsWith	Искомое значение(string) Правила сравнения(StringComparison) Игнорирование регистра(bool) Информация о языковых настройках (CultureInfo)	Метод возвращается логическое значение, определяющее, начинается или заканчивается строка заданной подстрокой.
PadLeft, PadRight	Нужная длина(int) Дополняющий символ (char)	Методы выполняют дополнение символами до заданной длины строки. Символы вставляются в начало или в конец строки. В итоге возвращается новая строка. Если символ не указан, используется пробел.

Имя метода	Параметры	Описание
Динамические методы		
Trim, TrimStart, TrimEnd	Нужная длина(int) Удаляемые символы (char[])	Удаляют символы в начале и в конце строки, или только с одного ее конца. В итоге возвращается новая строка.
ToCharArray	Стартовый индекс(int) Количество копируемых символов(int)	Метод преобразует строку или ее часть в массив символов.
Split	Разделитель (char, char[]) Максимальное число фрагментов(int) Правила разделения(StringSplitOptions)	Метод осуществляет разделение строки на элементы, используя разделители, и возвращая массив строк. Правила разделения позволяют игнорировать пустые элементы, которые могут быть получены, если в строке стояло подряд два разделителя.

Имя метода	Параметры	Описание
Статические методы		
Join	Разделитель (string) Коллекция фрагментов (string[], object[]...) Стартовый индекс(int) Количество копируемых символов(int)	Метод объединяет массив строк в единую строку. При этом между элементами массива вставляются разделители.
Concat	Коллекция или список строк(string[], object[]...)	Метод выполняет сцепление произвольного числа строк.

Класс StringBuilder

Данный класс используется в задачах обработки большого количества строк. Его особенность заключается в том, что он создает строковый буфер, за счет которого повышается производительность в указанных задачах. Кроме того, данный класс позволяет менять элементы строки напрямую, без создания новых строк. Например, для объекта класса StringBuilder допустимо выражение:

```
StringBuilder sb = new StringBuilder();
sb[0]='h' ;
```

Класс имеет несколько перегруженных конструкторов. Рассмотрим их подробнее:

```
StringBuilder sb = new StringBuilder(); //конструктор по умолчанию
StringBuilder sb1 = new StringBuilder(200); //начальный размер объекта
StringBuilder sb2 = new StringBuilder("gfgffgg"); //начальное значение
StringBuilder sb3 = new StringBuilder(20, 200); //начальная и максимальная емкости
```

```

        StringBuilder sb4 = new StringBuilder("dfdfdf", 200); /*
начальное значение и размер объекта */

        StringBuilder sb5 = new StringBuilder("gggg", 0, 4, 200); /*
объект с начальным размером 200 создается на основе 4х первых символов
указанной подстроки */

```

Как и любого класса, у `StringBuilder` определено множество методов и свойств. Ниже в таблице 5 приведено описание наиболее важных элементов.

Таблица 5. Методы и свойства класса `StringBuilder`

Имя метода или свойства	Описание
Length	Свойство задает текущий размер объекта <code>StringBuilder</code>
Capacity	Свойство задает максимально возможное количество символов, которое можно поместить в текущий объект
MaxCapacity	Возвращает максимальную емкость экземпляра
Insert	Метод вставляет подстроку в объект <code>StringBuilder</code> , начиная с заданной позиции
Clear	Удаляет все символы из объекта и обнуляет длину
Remove	Метод удаляет заданное количество символов из объекта, начиная с заданной позиции.
Replace	Метод заменяет все вхождения указанной подстроки новой подстрокой
Append, AppendFormat и AppendLine	Методы добавляют данные в конец объекта
EnsureCapacity	Метод используется для проверки емкости текущего объекта и если она меньше переданного в метод значения, то увеличивает эту емкость

Таким образом, мы рассмотрели базовый синтаксис языка C#, необходимый для разработки приложений анализа текстов. Теперь перейдем к рассмотрению конкретных алгоритмов. По ходу описания мы приведем необходимые уточнения по их реализации на языке C#, а затем рассмотрим пример разработки конкретного приложения для анализа текстов.

Контрольные вопросы к разделу

1. Перечислите классы в C#, работающие с текстовой информацией.
2. В чем отличие класса String от класса StringBuilder?
3. Какие методы класса String, на ваш взгляд, будут наиболее полезными при анализе текстов и почему?

Практические задания к разделу

1. Дана строка, содержащая несколько круглых скобок. Если скобки расставлены правильно (то есть каждой открывающей соответствует одна закрывающая), то вывести число 0. В противном случае вывести или номер позиции, в которой расположена первая ошибочная закрывающая скобка, или, если закрывающих скобок не хватает, число -1.
2. Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Определить количество слов, которые содержат ровно три буквы "А".
3. Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Определить длину самого длинного слова.
4. Дана строка-предложение на русском языке. Подсчитать количество содержащихся в строке гласных букв.
5. Дана строка-предложение на русском языке. Вывести самое длинное слово в предложении (если таких слов несколько, то вывести первое из них).

СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ

Как было сказано во введении, в последние десятилетия наблюдается неуклонный рост текстовой информации, нуждающейся в обработке. Причем, быстрой, эффективной и автоматизированной. Помимо обычного поиска по ключевым словам все чаще стали возникать задачи выделения данных из информационного ресурса, или его семантики, а также осуществление на основе нее каких-либо манипуляций. То есть, по сути, возникает задача автоматизированного анализа информационных ресурсов.

Первым шагом в анализе текстов всегда будет являться выделение из него значимых терминов. И большую роль в решении этой задачи играет составленный определенным образом словарь терминов. Чаще всего он составляется вручную экспертом после изучения определенного массива текстов по интересующей тематике. И понятно, что ручной труд в данном случае является трудоемким и ресурсозатратным. Поэтому в последнее время стали разрабатываться методы автоматизированного составления словарей.

В основе существующих алгоритмов извлечения терминов лежат статистические или лингвистические методы. Первые позволяют определить степень важности слова или словосочетания на основании определенных числовых закономерностей. Вторые же предполагают отбор по некоторым шаблонам, определенным для предметной области. Однако использование только этих методов или даже их комбинаций не позволяет учитывать особенности предметной области. Но эту проблему можно решить использованием семантических методов. Например, с применением кластерного или онтологического подходов.

Все методы обладают теми или иными достоинствами, недостатками, а также спецификой использования. Статистические методы отличаются универсальностью и могут применяться к различным языкам. Лингвистические же ограничены конкретным языком, но позволяют учесть его специфику. Кроме того, они более

сложные в реализации. Семантические методы позволяют учесть специфику предметной области, что повышает качество анализа. Однако они требуют определенных затрат от эксперта на создание базы знаний.

В основе статистических методов лежит представление о том, что термины – наиболее частотные слова или словосочетания, встречающиеся в текстах предметной области и обозначающие понятия предметной области. Если мы говорим о терминологических словосочетаниях (n-граммах), то они представляют собой двух-, трех-, четырехсловные сочетания, обладающие высокой степенью устойчивости.

Рассмотрим подробнее статистические методы работы с терминами текстов.

Статистический метод подсчета частот

Frequency – это метод прямого подсчета частоты n-словий. Результатом его работы является множество пар

$$F(x) = \{x, \text{count}_x\} \quad (1)$$

где x – n-слово, count_x – частота n-слова в рассматриваемом тексте.

Алгоритм метода включает в себя подсчет абсолютных частот всех n-словий целиком и упорядочивании списка. Метод предполагает, что n-слово – цельная конструкция, между частями которой допускаются лишь пробелы и дефисы. В основе метода лежит предположение, что высокочастотные n-слова являются значимыми понятиями.

Рассмотрим для примера самый простой случай: построение простого частотного словаря для всех слов текста. Тогда для реализации этого метода в C# можно использовать класс Dictionary.

Коллекция Dictionary – это ассоциативный массив, в котором доступ к элементам осуществляется не по индексу, а по так называемому ключу. Примером подобной коллекции может служить телефонная книга, в которой каждому значению присвоен не

порядковый номер, а смысловой идентификатор для удобства поиска элементов.

Синтаксис объявления Dictionary:

```
Dictionary<Тип_ключа, Тип_значения> имя;
```

Добавление элемента в Dictionary:

```
имя.Add(Ключ, Значение);
```

Необходимо заметить, что ключи должны быть уникальными.

Обращение к элементу с целью изменения его значения:

```
имя[ключ]=новое_значение;
```

Определение, есть ли значение с таким ключом в Dictionary:

```
имя.ContainsKey(ключ);
```

Рассмотрим пример добавления элемента в телефонную книгу, изменение его и определения, есть ли в ней номер Иванова:

```
Dictionary<string, string> phoneBook = new Dictionary<string, string>();  
phoneBook.Add("Петров", "+791929384");  
phoneBook["Петров"]="+78498953";  
if (phoneBook.ContainsKey("Иванов")) //действия, если да;  
else //действия, если нет;
```

Если же мы говорим об анализе текстов и методе подсчета частот, то ключом в данном случае будет термин, а значением – частота его встречаемости. Рассмотрим пример кода, реализующего данный метод. Итак, задача формулируется следующим образом: пусть в переменной input лежит некий текст, нам нужно построить его частотный словарь.

```
string input= «Ехал Грека через реку, видит Грека в реке рак.  
Сунул Грека руку в реку. Рак за руку Греку цап».
```

Первое, что нам необходимо сделать – выделить из текста слова. Для этого мы можем использовать метод Split класса String. Как мы помним, первым параметром он принимает массив разделителей. В данном тексте разделителями будут пробелы, точки и запятые. Так как иногда в тексте разделители стоят подряд, то в результате работы метода могут возникнуть так называемые пустые сущности. Чтобы этого избежать,

используем второй параметр, указывающий, что такие ситуации нужно игнорировать. И сохраняем все в итоге в массив строк `masSt`:

```
string[] masSt = input.Split(new char[] { ' ', '.', ',', '' },
StringSplitOptions.RemoveEmptyEntries);
```

Теперь создаем объект класса `Dictionary` со строковым ключом и числовым значением:

```
Dictionary<string, int> FreqDic = new Dictionary<string, int>();
```

Далее пробегаем по массиву строк и заполняем словарь:

```
foreach(string s in masSt)
    if (FreqDic.ContainsKey(s))
    {
        FreqDic[s]++;
    }
    else
        FreqDic.Add(s,1);
```

Таким образом, после работы данного кода в объекте `Dictionary` будет храниться следующая информация:

```
Ежал 1
Грека 3
через 1
реку 2
видит 1
в 2
реке 1
рак 1
Сунул 1
руку 2
Рак 1
за 1
Греку 1
цап 1
```

Как мы видим, метод работает не совсем корректно: например, `Рак` и `рак` были отнесены к разным словам из-за регистра. Подобная ситуация наблюдается со словами «реке» и «реку». Однако они были отнесены к разным словам за счет самой формы слова. Если первая

ситуация решается достаточно легко, то для решения второй нам необходимо будет производить морфологический анализ текста, но об этом будет сказано ниже, в отдельном разделе. Здесь же рассмотрим, как можно модифицировать код, чтобы убрать первую проблему. Самый простой способ – привести все сравниваемые слова к одному регистру. Например, к верхнему. Код в данном случае будет следующим:

```
foreach(string s in masSt)
    if (FreqDic.ContainsKey(s.ToUpper()))
        FreqDic[s.ToUpper()]++;
    else
        FreqDic.Add(s.ToUpper(), 1);
```

Итоговая же информация будет иметь вид:

```
ЕХАЛ 1
ГРЕКА 3
ЧЕРЕЗ 1
РЕКУ 2
ВИДИТ 1
В 2
РЕКЕ 1
РАК 2
СУНУЛ 1
РУКУ 2
ЗА 1
ГРЕКУ 1
ЦАП 1
```

Статистический метод Mutual Information

Это метод применяется только к двусловиям и предполагает вычисление коэффициента взаимной информации по формуле:

$$MI = \log_2 \frac{f(x, y) \times N}{f(x) \times f(y)} \quad (2)$$

где $f(x, y)$ – частота биграммы, $f(x)$, $f(y)$ – частота каждого слова в отдельности, N – количество слов в тексте.

Данный коэффициент показывает статистическую значимость встречаемости слов биграммы, сравнивая частоту их совместной встречаемости с произведением их относительных частот в тексте длины N. Значения коэффициента имеют следующий смысл. Если $1 < MI$, то сочетание статистически значимо. Если $0 < MI < 1$, то сочетание статистически незначимо. Значение $MI < 0$ говорит о том, что каждое из слов встречается лишь в тех позициях, в которых не встречается другое.

Модифицируем анализируемый текст следующим образом:

```
string input= «Ехал древний Грека через реку, видит древний Грека  
в реке древний рак. Сунул древний Грека руку в реку. Древний рак за  
руку древнего Греку цап».
```

Теперь проанализируем, является ли статистически значимым сочетание «древний Грека».

Первое, что нам нужно сделать: рассчитать частоту этой биграммы в анализируемом тексте. Сделать это можно по следующей формуле:

$$F_{xy}=(Count_all-Count_not_xy)/2 \quad (3)$$

Где count_all – общее число слов в тексте. Count_not_xy – число слов в тексте, из которого удалена нужная биграмма.

Чтобы удалить из текста бигramму, используем код, который замнет все ее вхождения на пустую строку:

```
string ModS = input.Replace("древний Грека", "");
```

Теперь также выгрузим все слова в массив, длина которого и даст нам искомую величину Count_not_xy. После этого мы можем легко рассчитать Fxy:

```
string[] masSt2 = ModS.Split(new char[] { ' ', '.', ',', '!' },  
StringSplitOptions.RemoveEmptyEntries);  
double Fxy=(masSt.Length-masSt2.Length)/2;
```

Далее, мы считаем, что в объекте FreqDic находится частотный словарь текста. Тогда, для получения частот каждого из слов биграммы, мы можем использовать код:

```
int fx=  
FreqDic.Where(x=>x.Key.Contains("Грека".ToUpper())).FirstOrDefault()  
.Value;
```

```
int fy =
FreqDic.Where(x=>x.Key.Contains("древний".ToUpper())).FirstOrDefault()
.Value;
```

Конструкция `x=>x....` – лямбда-выражение. Оно представляет собой специальный синтаксис для объявления анонимных функторов по месту их использования. Используя лямбда-выражения, можно объявлять функции в любом месте кода. Обычно лямбда-выражение допускает замыкание на лексический контекст, в котором это выражение использовано.

Лямбда-выражения принимают две формы. Форма, которая наиболее прямо заменяет анонимный метод, представляет собой блок кода, заключенный в фигурные скобки это — прямая замена анонимных методов. Лямбда-выражения, с другой стороны, предоставляют еще более сокращенный способ объявлять анонимный метод и не требуют ни кода в фигурных скобках, ни оператора `return`. Оба типа лямбда-выражений могут быть преобразованы в делегаты.

Во всех лямбда-выражениях используется лямбда-оператор `=>`, который читается как «переходит в». Левая часть лямбда-оператора определяет параметры ввода (если таковые имеются), а правая часть содержит выражение или блок оператора. Лямбда-выражение `x => x * 5` читается как «функция `x`, которая переходит в `x`, умноженное на 5»[3].

В нашем же примере лямбда-выражение читается как: «функция `x`, которая переходит в `x`, ключ которого содержит указанную подстроку». Затем мы отбираем из полученной коллекции первый и берем его значение. Так как ключи в `Dictionary` уникальны, то в итоге и так будет одно значение, но ограничения синтаксиса требуют дополнительную фильтрацию.

Теперь мы можем посчитать искомый коэффициент. Для этого используем код:

```
double MI= Math.Log((Fxy * masSt.Length)/(fx*fy), 2);
```

Для анализируемого текста $MI=2,3$, что говорит о статистической значимости.

Статистический метод T-Score

Это метод применяется также только к двусловиям. Он предназначен для определения степени взаимосвязи двух слов и предполагает вычисление коэффициента TS по формуле:

$$TS = \frac{f_{xy} - \frac{f_x f_y}{n}}{f_{xy}^2} \quad (4)$$

Где f_{xy} – частота биграммы, f_x , f_y – частота x и y соответственно, n – количество биграмм в тексте.

По сути, мера T-Score определяет вероятность, с которой можно утверждать, что между двумя словами имеется определенная связь. Причем, значение меры и частота двусловия в коллекции находятся в прямо-пропорциональной зависимости.

С точки зрения программной реализации, метод практически аналогичен рассмотренному выше, поэтому ее предлагается осуществить самостоятельно.

Статистический метод Log-Likelihood

Метод Log-Likelihood применяется только к двусловиям и рассчитывается по формуле:

$$LL = a * \log(a+1) + b * \log(b+1) + c * \log(c+1) + d * \log(d+1) - (a+b) * \log(a+b+1) - (a+c) * \log(a+c+1) - (b+d) * \log(b+d+1) - (c+d) * \log(c+d+1) + (a+b+c+d) * \log(a+b+c+d+1) \quad (5)$$

где a – частотность данной биграммы, b – суммарная частотность других (отличных от данной) биграмм с той же самой левой леммой, c – суммарная частотность других биграмм с той же самой правой леммой, d – суммарная частотность остальных биграмм текста.

Программная реализация метода отличается незначительно от рассмотренной выше, поэтому ее предлагается осуществить самостоятельно.

Статистический метод TF*IDF

Метод TF*IDF применяется только к однословиям и предполагает расчет двух коэффициентов. Первый из них – Term Frequency, то есть частота слова. Данный коэффициент представляет собой абсолютную частоту встречаемости данного слова относительно количества слов в текущем тексте, выбранном для анализа. Формула его расчета следующая:

$$TF = f(x) / N \quad (6)$$

где $f(x)$ – абсолютная частота слова в тексте, N – количество слов в тексте.

Второй коэффициент – Inverse Document Frequency, то есть обратная частота документа. Он представляет инверсию частоты встречаемости слова в документах коллекции или корпуса. Формула его расчета имеет вид:

$$TF * IDF(w) = TF * \log((n - b) / b) \quad (7)$$

где TF – это относительная частота слова в текущей коллекции, n – размер контрастной коллекции, b – число документов, в которых употреблялось слово w в контрастной коллекции.

В данном случае контрастная коллекция - это коллекция национального корпуса русского языка. Национальный корпус русского языка - это словарь, в котором имеется средняя частота употребления данного слова в любых текстах, художественных, научных и т.д. В нем указано относительная частота (берется мера 100 000 слов). То есть если в национальном корпусе стоит 1, то по статистике это слово встречается 1 раз на 100 000 слов во всем русском языке.

Под корпусом текстов, в данном случае понимается собрание текстов, обладающее некоторым набором базовых качеств [4]. К этим качествам относятся: репрезентативность; конечный размер; единство разметки; расположение на носителе информации; наличие процедуры отбора, использованной при составлении корпуса.

Программная реализация метода отличается незначительно от рассмотренной выше, поэтому ее предлагается осуществить самостоятельно.

Разработка приложения на языке C#

Рассмотрим пример разработки приложения для анализа текстов, основанного на приведенных методах. То есть написание полноценной программы на языке C#.

Программа на языке C# – это правильно построенная последовательность предложений. В общем случае программа представляет собой проект, содержащий 1..n файлов исходного кода на языке C#. Каждый из таких файлов содержит в себе объявления 1..m пространств имен, в каждом из которых определено 1..x классов. В любом проекте обязательно должен быть класс, содержащий специальный метод – точку входа. Поясним это понятие. Так как программа на языке C# может иметь множество классов со множеством методов, то необходимо каким-то образом определять точку, откуда начнет выполняться программа. Эта точка называется точкой входа и представляет собой метод любого класса, объявленный с заголовком `static void Main(string[] args)`. Точка входа может принадлежать любому классу, из описанных в программе, и она должна быть только одна.

В Visual Studio на языке C# можно создавать как консольные приложения, так и приложения Windows. Первый вариант хорош для тестирования написанных методов, так как не нужно тратить время на проработку интерфейса. Второй вариант предназначен для разработки удобных пользовательских приложений. Запуск любых приложений осуществляется командой меню Debug-Start debugging.

Отличие приложения Windows от консольного приложения заключается в том, что в рамках консольного приложения средством общения с пользователем была консоль. В приложении Windows с пользователем мы будем общаться посредством форм.

Для создания приложения Windows в Microsoft Visual Studio необходимо сделать следующее:

В меню «File» («Файл») выбрать пункт «New» («Новый») и в нем выбрать подпункт «Project» («Проект»). Далее в диалоговом окне, показанном в разделе Project types, необходимо выбрать пункт Visual C# и подпункт Windows. В разделе Templates выбрать Windows Forms Application. У нас на экране появилась основная форма (рисунок 2).

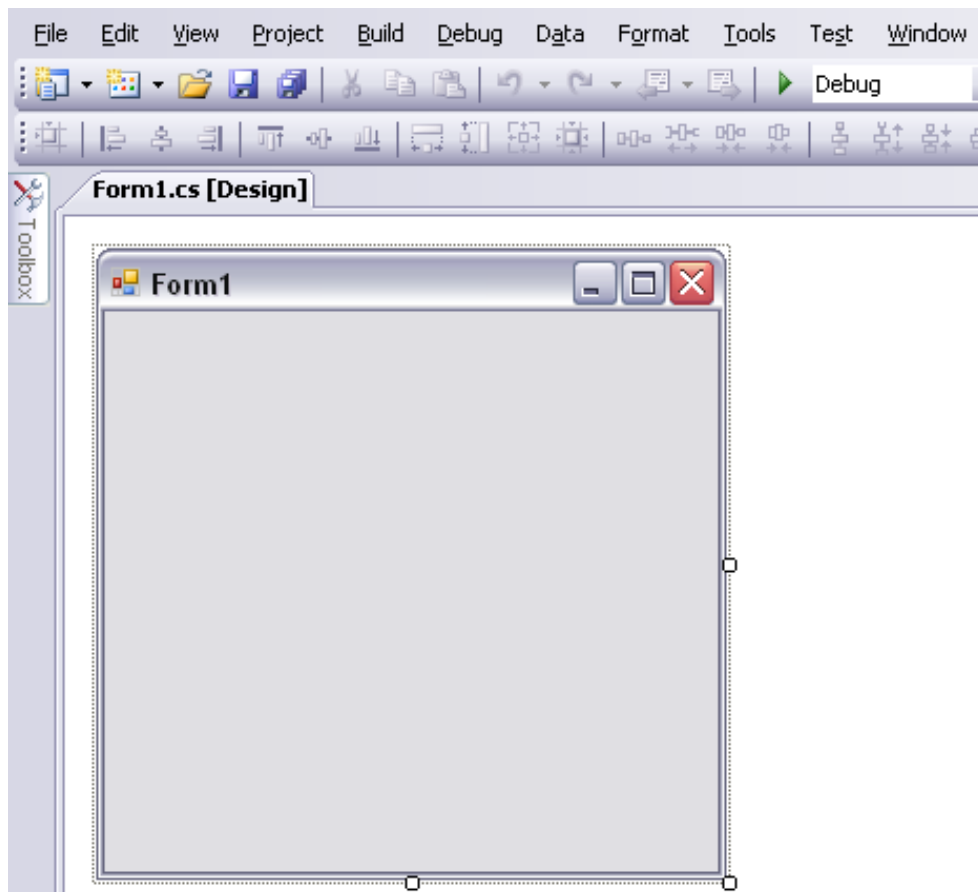


Рис. 2. Основная форма

Справа есть вкладка свойств Properties (если ее нет, нажимаем правой кнопкой на форму и выбираем пункт контекстного меню Properties). Задаем на ней заголовок формы в поле Text, например, «Анализ текстов» (рисунок 3).

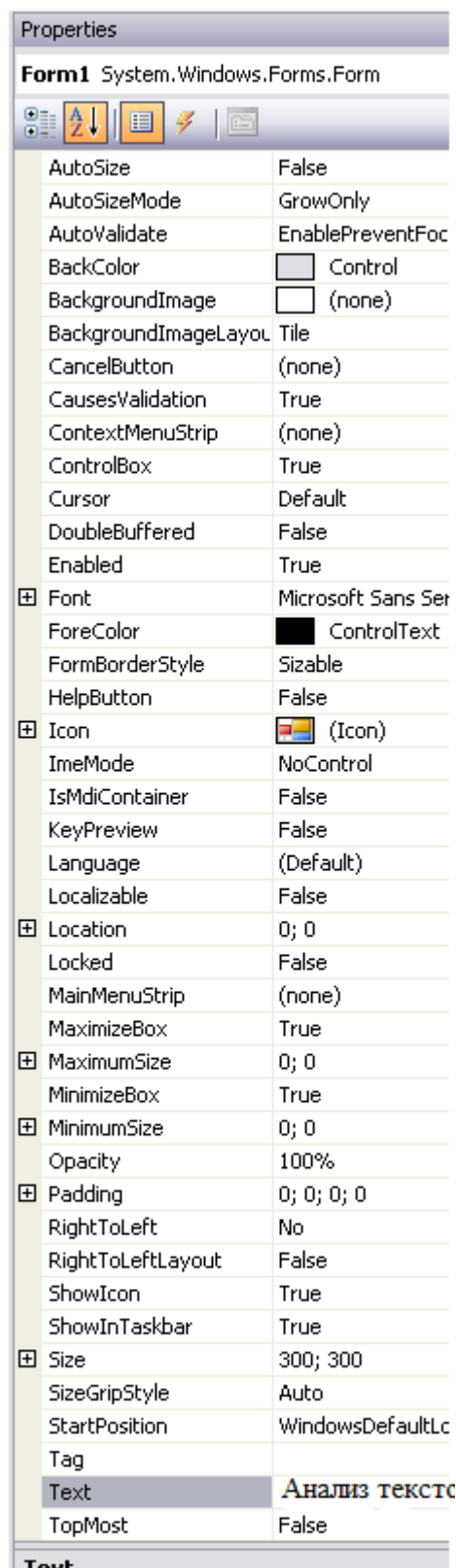


Рис. 3. Вкладка свойств

Для проектирования пользовательского интерфейса используются элементы, размещенные на вкладке Toolbox. Она расположена слева от формы. Если ее там нет, то идем в меню View-Toolbox. Когда мы

работаем с приложением WindowsForms, то внешний вид формы является неотъемлемой частью программы, отражая часть ее замысла. В связи с этим, в качестве примера рассмотрим разработку приложения, анализирующего текст с использованием метода подсчета частоты и коэффициента MI.

Пользовательский интерфейс можно спроектировать разными способами. Мы рассмотрим вариант, отличающийся достаточным уровнем удобства для пользователя и относительной простотой реализации для программиста.

Ввод текста для анализа логичнее всего осуществить с помощью текстового поля на форме. Вывод программы также будет выгружать в текстовое поле. Для этого выходим слева на вкладку Toolbox и выбираем в CommonControls элемент TextBox (рисунок 4).

Размещаем два элемента на форме или двойным щелчком мыши или перетаскиванием. Так как входная и выходная информация может быть большого объема, то в свойстве MultiLine у TextBox-ов ставим значение true. Примечание: для биграммы это необязательно.

Теперь меняем их свойства Name, чтобы в дальнейшем не запутаться, какой из них за что отвечает. Тогда текстовое поле с входной информацией будем называть tbInput, с выходной – tbOutPut, а поле с биграммой – tbBigram.

Для удобства пользователя разместим три элемента Label над текстовыми полями и подпишем, что каждое из них будет означать. Шрифты у элементов Label так же настраиваются в Properties (свойство Font), текст меняется там же (свойство Text). Обратите внимание, что вкладка Properties отображает свойства текущего (выделенного) элемента. Каждый из этих элементов имеет уникальное имя, по которому к нему можно обратиться из программного кода. Оно отображается жирным шрифтом вверху вкладки Properties.

Наша программа должна делать по сути две вещи: выводить простой частотный словарь введенного текста и делать вывод о

значимости какой-либо биграммы. Так как биграмму для анализа надо вводить в отдельном текстовом поле, то размещаем его на форме.

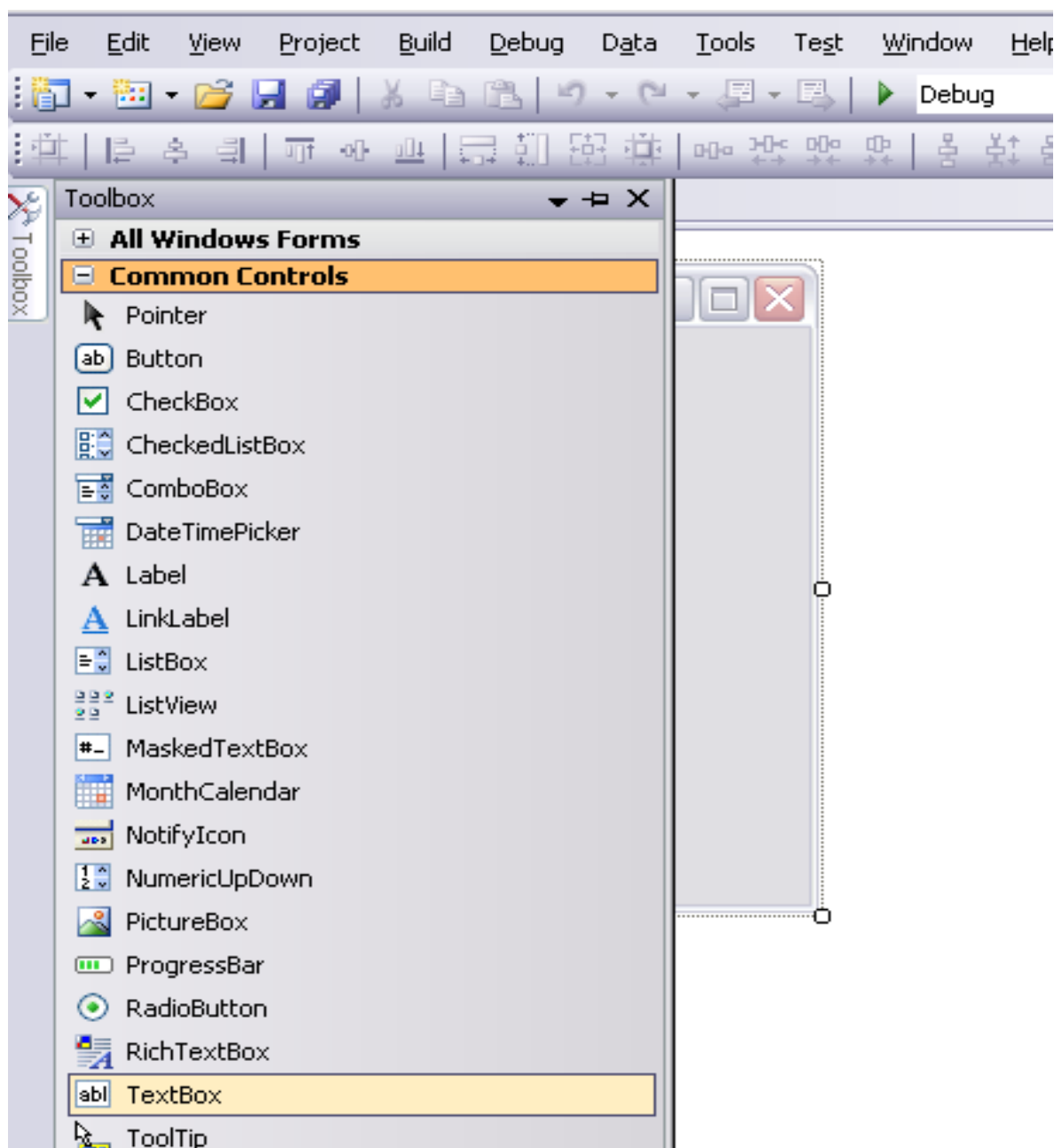


Рис. 4. Размещение текстовых полей

Для выполнения действий же размещаем на форме две кнопки, в их свойства Text записываем нужные заголовки, а в свойства Text текстовых полей сразу заносим контрольные примеры. Назовем первую кнопку bFreq, а вторую bMI.

В итоге внешний вид приложения может быть таким, как представлен на рисунке 5.

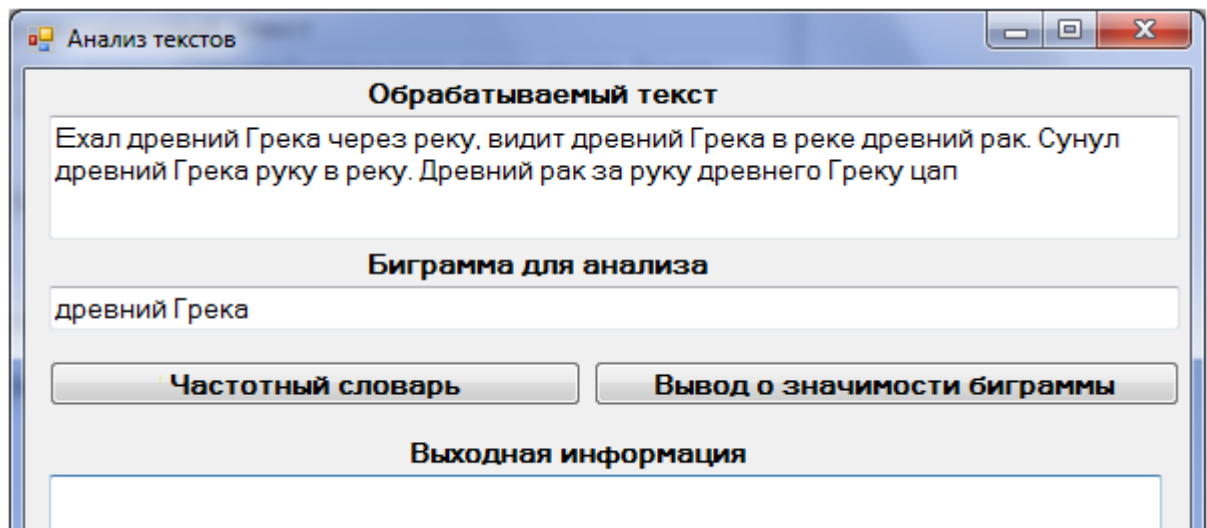


Рис. 5. Внешний вид программы

Двойным щелчком по первой кнопке вызываем редактор кода. При выполнении этого действия среда автоматически создала вам метод обработки нажатия на кнопку с пустым телом и в принципе, можно на позиции курсора написать код получения частотного словаря. Однако обратите внимание, что получение частотного словаря нам потребуется в обеих кнопках, поэтому мы вынесем его вместе с распечаткой в выходной TextBox в отдельный функциональный метод, определенный над методом обработки нажатия на кнопку:

```
public Dictionary<string, int> Method(string[] masSt)
{
    tbOutPut.Text = string.Empty;
    Dictionary<string, int> FreqDic = new Dictionary<string, int>();
    foreach (string s in masSt)
        if (FreqDic.ContainsKey(s.ToUpper()))
            FreqDic[s.ToUpper()]++;
        else
            FreqDic.Add(s.ToUpper(), 1);
    foreach (var k_v in FreqDic.OrderByDescending(x => x.Value))
    {
        tbOutPut.Text += k_v.Key + " " + k_v.Value + Environment.NewLine;
    }
    return FreqDic;
}
```

В данном случае метод OrderByDescending позволяет нам вывести словарь в порядке убывания частот. В самой же кнопке получения частотного словаря напомним код:

```
private void bFreq_Click(object sender, EventArgs e)
{
    string[] masSt = tbInput.Text.ToString().Split(new char[] { ' ',
    '.', ',', ',' }, StringSplitOptions.RemoveEmptyEntries);
    Dictionary<string, int> FreqDic = Method(masSt);
}
```

Во второй кнопке пишем следующий код:

```
private void bMI_Click(object sender, EventArgs e)
{
    string[] masSt = tbInput.Text.ToString().Split(new char[] { ' ',
    '.', ',', ',' }, StringSplitOptions.RemoveEmptyEntries);
    Dictionary<string, int> FreqDic = Method(masSt);
    string ModS =
    tbInput.Text.ToString().Replace(tbBigram.Text.Trim(), "");
    string[] masSt2 = ModS.Split(new char[] { ' ', '.', ',', ',' },
    StringSplitOptions.RemoveEmptyEntries);
    double fxy = (masSt.Length - masSt2.Length) / 2;
    string[] masBigr = tbBigram.Text.ToString().Split(new char[]
    { ' ' }, StringSplitOptions.RemoveEmptyEntries);
    int fx =
    FreqDic.Where(x =>
    x.Key.Contains(masBigr[0].ToUpper())) .FirstOrDefault().Value;
    int fy =
    FreqDic.Where(x =>
    x.Key.Contains(masBigr[1].ToUpper())) .FirstOrDefault().Value;
    double MI = Math.Log((fxy * masSt.Length) / (fx * fy), 2);
    tbOutPut.Text += Environment.NewLine + "fx:" + fx + " fy:" + fy;
    tbOutPut.Text += Environment.NewLine + "MI: " + MI;
    if (MI < 0)
        tbOutPut.Text += Environment.NewLine + "Каждое из слов
        встречается лишь в тех позициях, в которых не встречается другое";
    if (MI > 1) tbOutPut.Text += Environment.NewLine + "Значимо";
    if ((MI > 0) && (MI < 1)) tbOutPut.Text += Environment.NewLine +
    "Не значимо";
}
```

Результат работы программы представлен на рисунке 6.

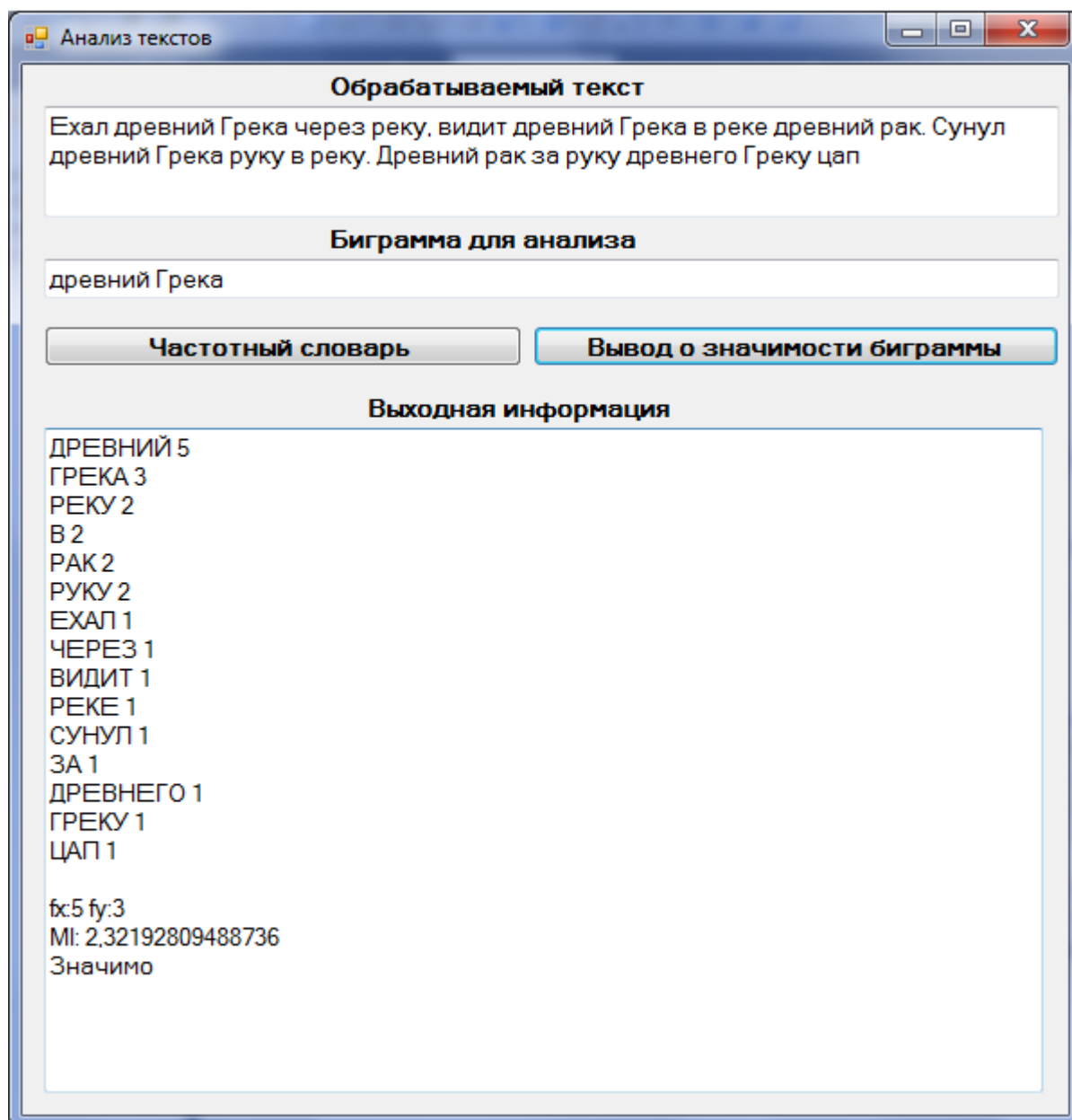


Рис. 6. Результат работы программы

Если мы внимательно посмотрим на частотный словарь, то увидим в нем два незначущих слова – предлоги «в» и «за». Они являются «мусором» и не несут никакой смысловой нагрузки. Для того чтобы они не влияли на результат, их необходимо исключить из рассматриваемого текста, прогнав его через так называемый стоп-лист, т.е. совокупность стоп-слов.

Рассмотрим определение этого термина, данное в Википедии:

«Стоп-слова или шумовые слова – термин из теории поиска информации по ключевым словам. Стоп-слова могут делиться на общие и зависимые. К общим можно отнести предлоги, суффиксы, причастия, междометия, цифры, частицы и т. п. Общие шумовые слова всегда исключаются из поискового запроса (за исключением поиска по строгому соответствию поисковой фразы), также они игнорируются при построении инвертированного индекса. Считается, что каждое из общих стоп-слов есть почти во всех документах коллекции.

Зависимые стоп-слова являются таковыми относительно семантики поисковой фразы. Идея заключается в том, чтобы по-разному учитывать отсутствие просто слов из запроса и зависимых стоп-слов из запроса в найденном документе.

Например, при поиске по запросу Пушкин Александр Сергеевич, есть смысл отобразить все документы содержащие:

Пушкин, Александр, Сергеевич

Пушкин, Александр

Пушкин, Сергеевич

Пушкин

Но вряд ли есть смысл отображать документы, содержащие только:

Александр, Сергеевич

Александр

Сергеевич

То есть в данном запросе шумовыми словами являются Александр и Сергеевич.

Зависимые стоп-слова отличаются тем, что в поисковом запросе их следует учитывать только при наличии в искомом документе значимых ключевых слов. К зависимым стоп-словам можно отнести Александр и Сергеевич из поискового запроса Пушкин Александр Сергеевич»[5].

Так как в нашей задаче стоп-лист небольшой, зададим его вручную:

```
string[] stopL = { "#в#", "#на#", "#по#", "#из#", "#за#" };
```


Обратите внимание на знаки решетки. Они необходимы, чтобы однозначно определять предлог от просто букв и слогов слова. Теперь проведем небольшую предобработку строки, заменив все разделители на знаки решетки, чтобы выделить предлоги, как записано в стоп-листе:

```
string ModS = tbInput.Text.ToString();
foreach(char c in ModS)
if (Char.IsPunctuation(c) || Char.IsSeparator(c))
    ModS=ModS.Replace(c, '#');
```

И теперь прогоним итоговый текст через стоп-лист, заменив предлоги новым разделителем:

```
foreach(string s in stopL)
    ModS = ModS.Replace(s, "#");
```

Обратите внимание: в предложенном варианте решения есть две неточности, которые предлагается обнаружить и устранить самостоятельно. Так как мы ввели новый разделитель текста, то в итоге код кнопки станет следующим:

```
private void bMI_Click(object sender, EventArgs e)
{
    string[] stopL = { "#в#", "#на#", "#по#", "#из#", "#за#" };
    string ModS = tbInput.Text.ToString();
    foreach(char c in ModS)
        if (Char.IsPunctuation(c) || Char.IsSeparator(c))
            ModS=ModS.Replace(c, '#');
    foreach(string s in stopL)
        ModS = ModS.Replace(s, "#");
    string[] masSt = ModS.Split(new char[] { '#' },
StringSplitOptions.RemoveEmptyEntries);
    Dictionary<string, int> FreqDic = Method(masSt);
    ModS = ModS.Replace(tbBigram.Text.Trim().Replace(' ', '#'), "");
    string[] masSt2 = ModS.Split(new char[] { '#' },
StringSplitOptions.RemoveEmptyEntries);
    double fxy = (masSt.Length - masSt2.Length) / 2;
    string[] masBigr = tbBigram.Text.ToString().Split(new char[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);
    int fx =
```

```

        FreqDic.Where(x
x.Key.Contains(masBigr[0].ToUpper()))).FirstOrDefault().Value;           =>
        int fy =
        FreqDic.Where(x
x.Key.Contains(masBigr[1].ToUpper()))).FirstOrDefault().Value;           =>
        double MI = Math.Log((fxy * masSt.Length) / (fx * fy), 2);
        tbOutPut.Text += Environment.NewLine + "fx:" + fx + " fy:" + fy;
        tbOutPut.Text += Environment.NewLine + "MI: " + MI;
        if (MI < 0) tbOutPut.Text += Environment.NewLine + "Каждое из
слов встречается лишь в тех позициях, в которых не встречается
другое";
        if (MI > 1) tbOutPut.Text += Environment.NewLine + "Значимо";
        if ((MI > 0) && (MI < 1)) tbOutPut.Text += Environment.NewLine + "Не
значимо";
    }

```

Теперь для наглядности построим графическое отображение частот.

Построение графиков средствами C#

Графики средствами C# можно построить различными способами: с помощью встроенных средств или с использованием сторонних библиотек. Мы рассмотрим построение графиков возможностями технологии GDI+.

GDI (Graphics Device Interface, Graphical Device Interface) является интерфейсом Windows, предназначенным для представления графических объектов и вывод их на монитор или принтер. На базе технологии GDI была разработана GDI+. Это улучшенная среда для 2D-графики, расширенная возможностями сглаживания линий, использования координат с плавающей точкой, градиентной заливки, использованием ARGB-цветов и т. п.

В данном пособии мы рассмотрим технику рисования в оперативной памяти с последующим выводом их на форму. Для рисования мы будем использовать объект класса Bitmap, а для вывода на форму – элемент управления PictureBox. Добавим в наш метод получения частотного словаря отображение его в графическом виде. Размещаем на форме PictureBox и переходим к нашему методу. В нем нам нужно, во-первых, создать объект класса Bitmap, с размерами,

совпадающими с нашим PictureBox. В данном случае Bitmap выполняет роль холста, а PictureBox – рамки, в которую этот холст повесят. Поэтому размеры должны совпадать. В итоге получаем код:

```
Bitmap bmp = new Bitmap(pictureBox1.Width, pictureBox1.Height);
```

Теперь мы должны создать объект класса Graphics – основного класса, предоставляющего доступ к возможностям GDI+. Для данного класса не определено ни одного конструктора. Его объект создается в ходе выполнения ряда методов применительно к конкретным объектам, у которых есть поверхность для рисования. Одним из таких объектов и является Bitmap. Поэтому создаем объект класса Graphics следующим образом:

```
Graphics gr = Graphics.FromImage(bmp);
```

Теперь все вызовы методов отображения фигур будут отрабатывать на нашей битовой карте. Класс Graphics содержит множество методов рисования вида Fill* или Draw*, отвечающих за отображение закрашенных или не закрашенных фигур. Первая группа методов в качестве одного из параметров принимает объект типа Brush (кисть), а вторая – объект типа Pen (карандаш). Исключение – метод DrawString, который отображает текст. Этот метод в качестве одного из параметров принимает объект Brush.

Для того чтобы график красиво отображался на форме, подстраиваясь под ее размер нам необходимо ввести коэффициенты перевода реальных координат в экранные. Для этого найдем значение максимального значения y и максимального значения x. Пусть эти координаты сохранены в переменные maxX и maxY соответственно:

```
int maxX = FreqDic.Values.Count();  
int maxY = FreqDic.Max(x => x.Value);
```

Тогда вводим два коэффициента перевода координат, рассчитываемых по формулам:

```
float cdx = (w - 90) / maxX;  
float cdy = (h - 50) / maxY;
```

В данном случае w – ширина `pictureBox`, а h – высота. 90 и 50 – отступы, необходимые, чтобы оси координат и подписи не упирались в края. Так как в GDI+ ось y идет из левого верхнего угла, то нам необходимо будет инвертировать координаты. Для этого введем дополнительную переменную

```
int max = h-25.
```

Нарисуем для начала оси координат. Для этого создадим переменную-карандаш, изображающую линии со стрелками на концах:

```
Pen p = new Pen(Brushes.Green, 3);  
p.EndCap = System.Drawing.Drawing2D.LineCap.ArrowAnchor;
```

Теперь, используя эту переменную, изобразим оси:

```
gr.DrawLine(p, (int)(0 * cdx)+25, (int)h-25, (int)(0 * cdx)+25, 0);  
gr.DrawLine(p, 0, (int)(max-0 * cdy), (int)w-25, (int)(max - 0 * cdy));
```

Как мы видим, здесь используется метод `DrawLine`. В качестве параметров он принимает переменную-карандаш и координаты $\{x, y\}$ начальной и $\{x, y\}$ конечной точек, которые нужно соединить линией. Координаты должны быть типа `int`, поэтому используем явное приведение к этому типу. Конкретно в этом случае приведение можно опустить, но чтобы в дальнейшем вывод осуществлялся по этому шаблону, мы намеренно его усложнили.

Теперь расчертим координатную сетку и выведем подписи по осям. Для оси x код будет следующим:

```
List<string> Ks = FreqDic.Keys.ToList<string>();  
for (int x = 0; x <maxX; x++) {  
    gr.DrawString(Ks[x].ToString(), new Font("Arial", 6),  
Brushes.Green, (int)(x * cdx) + 25, max + 10+((x%2==0)?7:0));  
    gr.DrawLine(Pens.Green, (int)(x * cdx) + 25, (int)h - 25,  
(int)(x * cdx) + 25, 0); }  

```

Для оси ординат же код будет таким:

```
for (float y = 1; y <= maxY + 1; y++) {  
    gr.DrawString((y).ToString(), new Font("Arial", 10),  
Brushes.Green, 0, (int)(max - (y) * cdy));  
    gr.DrawLine(Pens.Green, 0, (int)(max - (int) (y * cdy)), (int)w  
- 25, (int)(max - (int) (y * cdy))); }  

```

Обратите внимание, коллекция ключей в Dictionary хранится в свойстве Keys, но для доступа к ней и перечислению нам необходимо выгрузить ее в объект типа List<string>. Выражение $(x \% 2 == 0) ? 7 : 0$ обеспечивает написание слов в две строчки для красоты графика. Однако оно подобрано конкретно под эту задачу и в других условиях может не дать нужного эффекта.

Теперь займемся отображением графика. Частоты логичнее представлять в виде гистограммы. Поэтому получим код:

```
for (int i = 0; i < Ks.Count(); i++)
    gr.FillRectangle(Brushes.Red, (int)(i * cdx) + 25, (int)(max - FreqDic[Ks[i]] * cdy), 10, (int)(FreqDic[Ks[i]] * cdy));
```

Если вы сейчас запустите приложение и нажмете на кнопку, то ничего не произойдет. Вернее, график будет нарисован, но мы его не увидим, так как сейчас он находится в оперативной памяти. Для того чтобы отобразить нарисованную картинку на форме, необходимо добавить следующую строчку кода:

```
pictureBox1.Image = bmp;
```

В итоге если вы нажмете на кнопку простого получения частотного словаря, у вас получится примерно следующее (рисунок 7):

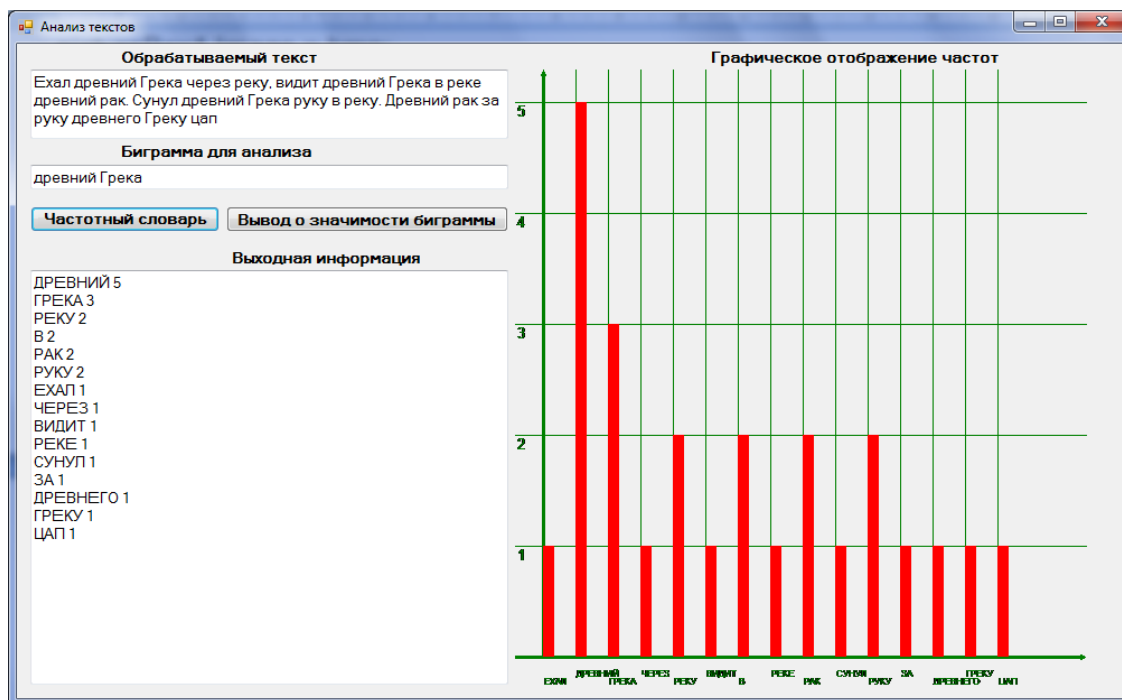


Рис. 7. Отображение гистограммы

Теперь усовершенствуем наше приложение, добавив ему функцию загрузки анализируемого текста из файла.

Работа с файлами

Для работы с текстовыми файлами вам понадобятся классы `File`, `StreamReader` или `StreamWriter`. Основные классы для работы с файлами располагаются в пространстве имен `System.IO`. Рассмотрим примеры записи и чтения текстовых файлов.

```
//Не забудьте подключить следующий namespace:
using System.IO;
//Запись в файл:
string Filename = "write.txt";
using (StreamWriter wr = File.CreateText(Filename))
{
    wr.WriteLine("Записываемый текст");
}
//Чтение всего текста из файла:
string Filename2 = "read.txt";
string s=File.ReadAllText(Filename2);
//Чтение всего текста из файла построчно:
string[] s= File.ReadAllLines(Filename2);
```

Чтобы дать пользователю возможность выбрать файл, в приложениях `WindowsForms` есть классы `OpenFileDialog` и `SaveFileDialog`. Основной их метод – `ShowDialog`. Он возвращает значение типа `DialogResult.OK`, если пользователь осуществил свой выбор. Имя выбранной папки будет храниться в свойстве `SelectedPath`, а выбранного файла – в свойстве `FileName`.

Так как загрузка из файла, не единственная модификация, которую мы будем делать, то добавим в наше приложение главное меню. На панели инструментов это компонент `MenuStrip`. Сам компонент разместится под формой, а на ней в самом верху появится настраиваемая строка меню. Для его заполнения просто переходите на нужные кнопки, помеченные подсказкой «Type here» (рисунок 8):

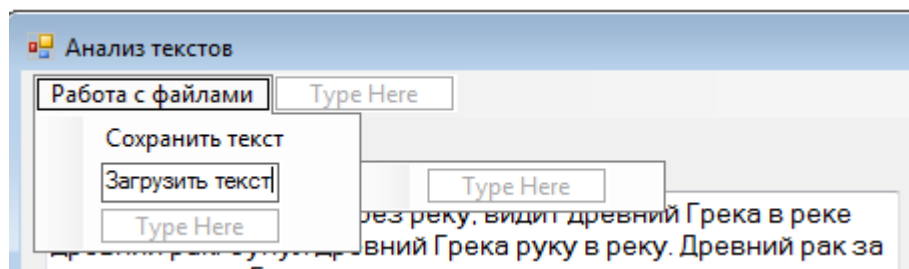


Рис. 8. Создание меню

Мы добавим две функции: сохранение текста, записанного в textBox и загрузка в него текста из файла. Для привязки кода к кнопкам меню щелкаем дважды по соответствующему пункту. В полученном методе пишем следующий код:

```
private void сохранитьТекстToolStripMenuItem_Click(object sender,
EventArgs e)
{
    try
    {
        SaveFileDialog sv = new SaveFileDialog();
        sv.Filter = "Текстовые файлы|*.txt";
        if (sv.ShowDialog() == DialogResult.OK)
        {
            using (StreamWriter wr = File.CreateText(sv.FileName))
            {
                wr.WriteLine(tbInput.Text);
            }
        }
        MessageBox.Show("Успешно!!");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Сделаем несколько пояснений. Конструкция try-catch в данном случае нужна для так называемой обработки исключений. Например, если программа попытается записать информацию на переполненный диск, то будет выдано корректное сообщение и программа не потеряет своей работоспособности. Строчка sv.Filter = «Текстовые файлы|*.txt»;

обеспечит сохранение файла в формате txt. А конструкция `MessageBox.Show` выдаст сообщение о результате работы.

Загрузка из файла осуществляется с незначительными изменениями:

```
private void загрузитьТекстToolStripMenuItem_Click(object sender,
EventArgs e)
{
    try
    {
        OpenFileDialog od = new OpenFileDialog();
        od.Filter = "Текстовые файлы|*.txt";
        if (od.ShowDialog() == DialogResult.OK)
        {
            tbInput.Text= File.ReadAllText(od.FileName);
        }
        MessageBox.Show("Успешно!!");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Обратите внимание, если вас интересуют не только текстовые файлы, то строчку, задающую фильтр, можно закомментировать.

Теперь расширим наш функционал, добавив в него возможность морфологического анализа полученного частотного словаря с целью его более правильного составления. Для этого рассмотрим работы с морфологическим анализатором `mystem`.

Контрольные вопросы к разделу

1. Какие существуют основные статистические методы извлечения терминологии из текстов? Какие из них используются только для анализа двусловий?
2. Что такое «лямбда-выражения» и для чего они нужны?

3. Что такое «стоп-слова»?

Практические задания к разделу

1. Проведите статистический анализ любого технического текста, содержащего не менее 250 слов. Постройте его частотный словарь.
2. Проведите статистический анализ любого технического текста, содержащего не менее 250 слов. Выявите наиболее статистически значимые двусловия по методу Mutual Information.
3. Проведите статистический анализ любого технического текста, содержащего не менее 250 слов. Выявите наиболее статистически значимые двусловия по методу T-Score.
4. Проведите статистический анализ любого технического текста, содержащего не менее 250 слов. Выявите наиболее статистически значимые двусловия по методу Log-Likelihood
5. Проведите статистический анализ любого технического текста, содержащего не менее 250 слов. Выявите наиболее статистически значимые двусловия по методу $TF \times IDF$.

МОРФОЛОГИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ

В русском языке десять частей речи: имя существительное, глагол, имя прилагательное, имя числительное, наречие, местоимение, предлог, союз, частица, междометие. Определение принадлежности заданного слова к одной из них называется морфологическим анализом. При выделении терминов этот шаг является крайне необходимым, так как необходимо учитывать, что слова могут употребляться в различных падежах, склонениях, временах. Следовательно, для правильной интерпретации слова необходимо привести его к канонической, основной форме. То есть, выделить его лемму.

Программа «Mystem» представляет собой морфологический анализатор русского языка с поддержкой снятия морфологической неоднозначности. Он был разработан Ильей Сегаловичем в компании «Яндекс». Программа работает на основе словаря и может формировать морфологические гипотезы о неизвестных ей словах.

Программа представляет собой бесплатно распространяемое консольное приложение. По умолчанию mystem работает с UTF-8 и настроено на стандартный ввод-вывод (клавиатура-экран). Если же вам необходимо работать с информацией из файла и выгружать результат туда же, то рекомендуется запускать приложение из командной строки. Формат вызова следующий:

```
mystem.exe [опции работы] [имя входного файла] [имя выходного файла]
```

Например, пусть mystem.exe лежит на диске D, в корне. Там же лежит входной файл input.txt и туда же надо сохранить файл output.txt, тогда после вызова командной строки (например, командой cmd) команда вызова mystem должна быть следующей:

```
C:\> D:\mystem.exe D:\input.txt D:\output.txt
```

Если какую-то из опций вызова нужно опустить, то вместо нее ставится прочерк.

Ниже в таблице 6, взятой из официальной документации [6], приведен список возможных опций с расшифровкой.

Таблица 6. Опции mystem

Символ опции	Описание
-n	Построчный режим; каждое слово печатается на новой строке.
-c	Копировать весь ввод на вывод. То есть, не только слова, но и межсловные промежутки. Опция необходима для возврата к полному представлению текста. В случае построчного вывода (когда задана опция n) межсловные промежутки вытягиваются в одну строку, символы перевода строки заменяются на \r и/или \n. Пробел для большей видимости заменяется подчеркиванием. Символ \ заменяется \\, подчеркивание заменяется _. Таким образом можно однозначно восстановить исходный текст.
-w	Печатать только словарные слова.
-l	Не печатать исходные словоформы, только леммы и граммеы.
-i	Печатать грамматическую информацию, расшифровка ниже.
-g	Склеивать информацию словоформ при одной лемме (только при включенной опции -i).

Символ опции	Описание
-s	Печатать маркер конца предложения (только при включенной опции -c).
-e	Кодировка ввода/вывода. Возможные варианты: cp866, cp1251, koi8-r, utf-8 (по умолчанию).
-d	Применить контекстное снятие омонимии.
--eng-gr	Печатать английские обозначения граммов.
--filter-gram	Строить разборы только с указанными граммами.
--fixlist	Использовать файл с пользовательским словарём.
--format	Формат вывода. Возможные варианты: text, xml, json. Значение по умолчанию – text.
--generate-all	Генерировать все возможные гипотезы для несловарных слов.
--weight	Печатать бесконтекстную вероятность леммы.

Приведем пример работы с программой. Пусть в файле G:\1.txt лежит текст: «Копировать весь ввод на вывод. То есть, не только слова, но и межсловные промежутки. Опция необходима для возврата к полному представлению текста». Мы хотим получить его анализ в файле G:\2.txt. Обратите внимание, исходный файл должен быть сохранен в кодировке utf-8.

При простом запуске программы

```
C:\> G:\mystem.exe G:\1.txt G:\2.txt
```

в выходном файле получим следующий текст:

«Копировать { копировать } весь { весь | весить } ввод { ввод } на { на } вывод { вывод } То { то | тот } есть { быть | есть } не { не } только { только } слова { слово } но { но } и { и } межсловные { межсловный? | межсловная? } промежутки { промежуток } Опция { опция } необходима { необходимый } для { для } возврата { возврат } к { к } полному { полный } представлению { представление } текста { текст }»

Как мы видим, после каждого слова в фигурных скобках указана его лемма. Если однозначно определить ее нельзя, то предложены возможные варианты.

Если же мы запустим программу с параметром -i

```
C:\> G:\mystem.exe -i G:\1.txt G:\2.txt
```

то получим развернутую информацию:

«Копировать { копировать=V, несов, пе=инф } весь { весь=APRO=вин, ед, муж, неод=APRO=им, ед, муж | весь=S, жен, неод=вин, ед | =S, жен, неод=им, ед | весить=V, несов, пе=ед, пов, 2-л } ввод { ввод=S, муж, неод=вин, ед | =S, муж, неод=им, ед } на { на=PR= | на=PART= } вывод { вывод=S, муж, неод=вин, ед | =S, муж, неод=им, ед } То { то=SPRO, ед, сред, неод=вин | =SPRO, ед, сред, неод=им | то=CONJ= | то=PART= | тот=APRO=вин, ед, сред | =APRO=им, ед, сред } есть { быть=V, нп=непрош, ед, изъяс, 3-л, несов | =V, нп=непрош, мн, изъяс, 3-л, несов | =V, нп=наст, мн, изъяс, 1-л, несов | =V, нп=наст, ед, изъяс, 1-л, несов | =V, нп=наст, мн, изъяс, 2-л, несов | =V, нп=наст, ед, изъяс, 2-л, несов | =V, нп=наст, мн, изъяс, 3-л, несов | =V, нп=наст, ед, изъяс, 3-л } не { не=PART= } только { только=PART= | только=CONJ= | только=ADV= } слова { слово=S, сред, неод=вин, мн | =S, сред, неод=род, ед | =S, сред, неод=им, мн } но { но=CONJ= | но=INTJ= } и { и=CONJ= | и=INTJ= | и=S, сокp=пр, мн | =S, сокp=пр, ед | =S, сокp=вин, мн | =S, сокp=вин, ед | =S, сокp=дат, мн | =S, сокp=дат, ед | =S, сокp=род, мн | =S, сокp=род, ед | =S, сокp=твор, мн | =S, сокp=твор, ед | =S, сокp=им, мн | =S, сокp=им, ед | и=PART= } межсловные { межсловный?=A=вин, мн, полн, неод | ?=A=им, мн, полн | межсловный?=A=вин, мн, полн, неод | ?=A=им, мн, полн | межсловная?=S, жен, неод=вин, мн | ?=S, жен, неод=им, мн | межсловный?=A, полн=им, мн | ?=A, полн=вин, мн, неод } промежутки { промежуток=S, муж, неод=вин, мн | =S, муж, неод=им, мн } Опция { опция=S, жен, неод=им, ед } необходима { необходимый=A=ед, кр, жен } для { для=PR= } возврата { возврат=S, муж, неод=род, ед } к { к=PR= | к=S, сокp=пр, мн | =S, сокp=пр, ед | =S, сокp=вин, мн | =S, сокp=вин, ед | =S, сокp=дат, мн | =S, сокp=дат, ед | =S, сокp=род, мн | =S, сокp=род, ед | =S, сокp=твор, мн | =S, сокp=твор, ед | =S, сокp=им, мн | =S, сокp=им, ед } полному { полный=A=дат, ед, полн, муж | =A=дат, ед, полн, сред } представлению { представление=S, сред, неод=дат, ед } текста { текст=S, муж, неод=род, ед }»

Для понимания этой информации рассмотрим расшифровку сокращений, приведенную в таблице 7, взятой из официальной документации [6]:

Таблица 7. Расшифровка условных обозначений

Кириллица	Латиница	Расшифровка
Части речи		
-	A	прилагательное
-	ADV	наречие
-	ADVPRO	местоименное наречие
-	ANUM	числительное-прилагательное
-	APRO	местоимение-прилагательное
-	COM	часть композита - сложного слова
-	CONJ	союз
-	INTJ	междометие
-	NUM	числительное
-	PART	частица
-	PR	предлог
-	S	существительное
-	SPRO	местоимение-существительное
-	V	глагол
Падеж		
им	nom	именительный
род	gen	родительный
дат	dat	дательный
вин	acc	винительный
твор	ins	творительный
пр	abl	предложный
парт	part	партитив (второй родительный)
местн	loc	местный (второй предложный)
зват	voc	звательный

Кириллица	Латиница	Расшифровка
Время (глаголов)		
наст	praes	настоящее
непрош	inpraes	непрошедшее
прош	praet	прошедшее
деепр	ger	деепричастие
инф	inf	инфинитив
прич	partcp	причастие
изъяв	indic	изъявительное наклонение
пов	imper	повелительное наклонение
Форма прилагательных		
кр	brev	краткая форма
полн	plen	полная форма
притяж	poss	притяжательные прилагательные
Степень сравнения		
прев	supr	превосходная
срав	comp	сравнительная
Лицо глагола		
1-л	1p	1-е лицо
2-л	2p	2-е лицо
3-л	3p	3-е лицо
Род		
муж	m	мужской род
жен	f	женский род
сред	n	средний род
Вид		
несов	ipf	несовершенный
сов	pf	совершенный

Кириллица	Латиница	Расшифровка
Залог		
действ	act	действительный залог
страд	pass	страдательный залог
Одушевленность		
од	anim	одушевленное
неод	inan	неодушевленное
Переходность		
пе	tran	переходный глагол
нп	intr	непереходный глагол
Прочие обозначения		
вводн	parenth	вводное слово
гео	geo	географическое название
затр	awkw	образование формы затруднено
имя	persn	имя собственное
искаж	dist	искаженная форма
мж	mf	общая форма мужского и женского рода
обсц	obsc	обсценная лексика
отч	patrн	отчество
прдк	praed	предикатив
разг	inform	разговорная форма
редк	rare	редко встречающееся слово
сокр	abbr	сокращение
устар	obsol	устаревшая форма
фам	famn	фамилия

Дополним теперь наше приложение функцией морфологического анализа. Пусть работать она будет следующим образом: у нас будет файл с текстом для анализа и программа `mystem`. Задача нашей программы: вызвать `mystem`, передав ей входной файл и, получив от нее

файл с выделенными леммами, привести каждое слово исходного текста в каноническую форму (то есть найти его лемму). То есть, из текста: «Ехал древний Грека через реку, видит древний Грека в реке древний рак. Сунул древний Грека руку в реку. Древний рак за руку древнего Греку цап» должен получиться текст: «Ехать древний Грек через река, видеть древний Грек в река древний рак. Сунуть древний Грек рука в река. Древний рак за рука древний Грек цапнуть». Этот процесс называется лемматизацией. И его нужно отличать от стемминга. Стеемминг – это процесс нахождения основы слова для заданного исходного слова. Основа слова необязательно совпадает с морфологическим корнем слова и не обязательно является его канонической формой.

В принципе, последовательность действий можно организовать по-разному. В данном случае предлагается следующее: на кнопку меню повесить выбор файла с исходной информацией, затем выбор файла для сохранения, выбор файла программы Mystem и, наконец, выгрузку преобразованного текста в textBox.

Для выбора первого файла используем объект OpenFileDialog, настроенный для удобства пользователя следующим образом. Во-первых, в его заголовке должен стоять текст «Выберите файл с текстом для анализа», а во-вторых, он должен быть настроен только на текстовые файлы. Выполняется это следующим кодом:

```
OpenFileDialog od = new OpenFileDialog();  
od.Filter = "Текстовые файлы|*.txt";  
od.Title = "Выберите файл с текстом для анализа";
```

Теперь запускаем его и, при нажатии клавиши ОК, выполняем выбор файла для сохранения. Для этого мы используем SaveFileDiaog со схожими параметрами. После его успешного запуска, мы должны выбрать файл с программой mystem. Для этого мы используем уже созданный ранее OpenFileDialog, предварительно сохранив имя

выбранного ранее файла в отдельную переменную, а также перенастроив его фильтр на исполняемые файлы.

Теперь мы должны запустить программу `mystem`, передав ей имена файлов. Для этого мы используем объект класса `System.Diagnostics.Process`. Его свойство `StartInfo`, содержит поля `FileName` и `Arguments`. В первое передаем путь к файлу `mystem.exe`, во второе – имена входного и выходного файла, разделенные пробелами. Для запуска приложения вызываем метод `Start`. После этого в итоговом файле мы получим следующий текст:

```
«Ехал{ехать} древний{древний} Грека{грек} через{через} реку{река} види  
т{видеть} древний{древний} Грека{грек} в{в} реке{река} древний{древний} рак{  
рак|рака} Сунул{сунуть} древний{древний} Грека{грек} руку{рука} в{в} реку{ре  
ка} Древний{древний} рак{рак|рака} за{за} руку{рука} древнего{древний} Греку  
{грек} цап{цап}».
```

На этом этапе мы должны выделить из него леммы. Эту задачу можно также решить несколькими способами. Рассмотрим одно из возможных решений. Разобьем полученный текст на пары слово-лемма. Для этого используем метод `Split` с разделителем «`}`». В итоге получаем следующий массив строк:

```
Ехал{ехать  
древний{древний  
Грека{грек  
через{через  
реку{река  
видит{видеть  
древний{древний  
Грека{грек  
в{в  
реке{река  
древний{древний  
рак{рак|рака  
Сунул{сунуть  
древний{древний  
Грека{грек  
руку{рука  
в{в
```

```

реку{река
Древний{древний
рак{рак|рака
за{за
руку{рука
древнего{древний
Греку{грек
цап{цап

```

Обратите внимание, что глагол «цапнуть» не опознался, в силу отсутствия специфического окончания. Но в данном примере это существенной роли не сыграет.

Еще нам необходимо учесть, что слово «рак» у нас определилось неоднозначно: для его леммы предлагается вариант «рак|рака» (мужской и женский род). Так как в нашем случае лемма является лишь идентификатором слова, то нам подойдет любой из вариантов. Поэтому при финальной обработке мы будем ориентироваться на первую выданную программой лемму. Таким образом, для финальной обработки мы организуем проход по полученному массиву, разделяя каждый его элемент по знаку «{» и удаляя при необходимости последовательность символов «|слово». В итоге код обработки нажатия на кнопку меню будет иметь следующий вид:

```

private void морфологическийАнализToolStripMenuItem_Click(object
sender, EventArgs e)
{
    try
    {
        string inpf;
        OpenFileDialog od = new OpenFileDialog();
        od.Filter = "Текстовые файлы|*.txt";
        od.Title = "Выберите файл с текстом для анализа";
        if (od.ShowDialog() == DialogResult.OK)
        {
            inpf = od.FileName;
            SaveFileDialog sv = new SaveFileDialog();
            sv.Filter = "Текстовые файлы|*.txt";

```

```

        sv.Title = "Выберите файл для сохранения результата";
        if (sv.ShowDialog() == DialogResult.OK)
        {
            od.Title = "Выберите файл программы mystem";
            od.Filter = "Приложения|*.exe";
            if (od.ShowDialog() == DialogResult.OK)
            {
System.Diagnostics.Process command = new System.Diagnostics.Process();
                command.StartInfo.FileName = od.FileName;
                command.StartInfo.Arguments = inpf + " " + sv.FileName;
                command.Start();

MessageBox.Show("Успешно!!");

                tbInput.Text = File.ReadAllText(sv.FileName);
                tbOutPut.Text = string.Empty;
                string[] Mas = tbInput.Text.Split(new char[] { '|' },
                StringSplitOptions.RemoveEmptyEntries);
                string ItogS = string.Empty;
                foreach (string s in Mas)
                {
                    if (s.Contains('|'))
                        tbOutPut.Text += s.Remove(s.IndexOf('|')) + Environment.NewLine;
                    else
                        tbOutPut.Text += s + Environment.NewLine;
                    string[] st=s.Split('{');
                    if (st[1].Contains('|'))
                        ItogS += st[1].Remove(st[1].IndexOf('|')) + " ";
                    else
                        ItogS += st[1] + " "; }
                tbInput.Text = ItogS; }
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
}

```

Итоговый вид программы и результат ее работы приведен на рисунке 9.

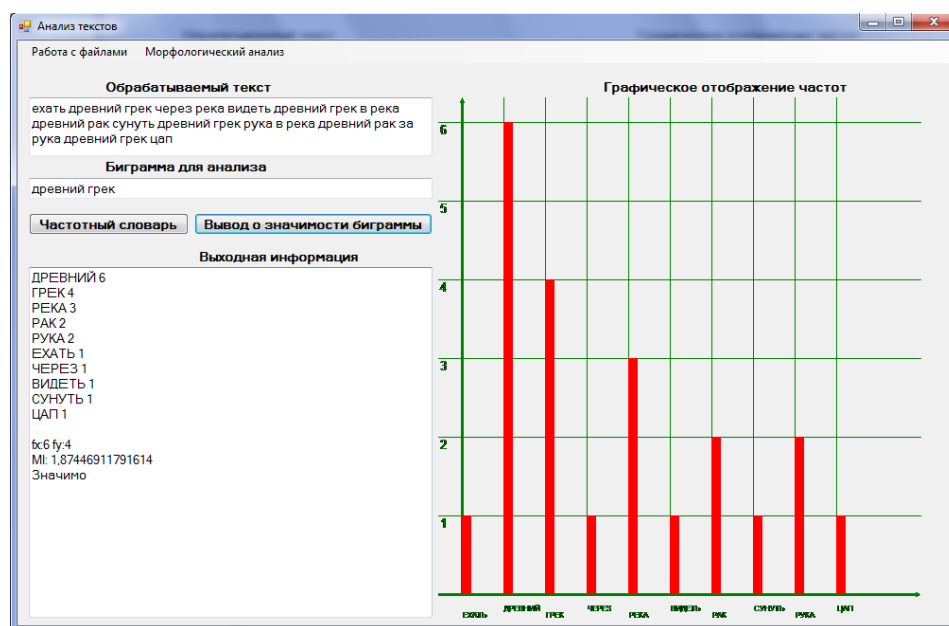


Рис. 9. Итоговый вид программы

Обратите внимание, что в этом примере биграмма для анализа вводится как «древний грек».

Мы рассмотрели статистический и морфологический способы анализа текстов. Теперь перейдем к кластерному.

Контрольные вопросы к разделу

1. Что такое морфологический анализ слова?
2. Что такое лемматизация?
3. Чем лемматизация отличается от стемминга?

Практические задания к разделу

С помощью программы Mystem проведите подробный морфологический анализ любого технического текста, содержащего не менее 250 слов. Программно определите количество:

1. Глаголов.
2. Существительных женского рода.
3. Прилагательных.
4. Существительных, стоящих во множественном числе.
5. Глаголов, стоящих в прошедшем времени.

КЛАСТЕРНЫЙ АНАЛИЗ ТЕКСТОВ

При анализе текстов у нас может возникнуть следующая задача: есть список слов или словосочетаний и необходимо определить, какие из них являются для текущего текста терминами. Эту задачу мы можем решить с помощью нечеткой кластеризации, определив два кластера: «термин» и «нетермин». Тогда по какой-либо статистической характеристике термины могут быть отнесены с той или иной степенью принадлежности к одному из типов.

Для нечеткого разбиения некоего множества точек на заданное количество кластеров можно использовать fcm-алгоритм кластеризации. Целью FCM (Fuzzy Classifier Means) – алгоритма кластеризации является автоматическая классификация множества объектов, которые задаются векторами признаков в пространстве признаков. FCM – алгоритм предполагает, что объекты принадлежат всем кластерам с определенной степенью принадлежности. Она определяется расстоянием от объекта до соответствующих кластерных центров. Данный алгоритм итерационно вычисляет центры кластеров и новые степени принадлежности объектов и основан на минимизации целевой функции по мере расстояния объектов от центров кластеров:

$$J = \sum_{i=1}^N \sum_{j=1}^C (u_{ij})^m \|x_i - c_j\| \quad (8)$$

где N - количество объектов,

C - количество кластеров,

u_{ij} - степень принадлежности объекта i кластеру j ,

m – любое действительное число, большее 1,

x_i - i -ый объект набора объектов,

c_j - j -ый кластер набора кластеров,

$\|x_i - c_j\|$ - норма, характеризующая расстояние от центра кластера j до объекта i .

Объектами кластеризации в данном нашем случае являются элементы слова. Вектор признаков объектов кластеризации в данном случае содержит только значение какой-либо статистической метрики (например, частоты).

Алгоритм нечеткой кластеризации выполняется по шагам. Рассмотрим каждый из шагов подробнее, приведя необходимые модификации целевой функции для упрощения дальнейшего программирования.

Первый шаг – инициализация. На этом шаге задаются параметры кластеризации, и инициализируется первоначальная матрица принадлежности объектов кластерам. К параметрам относятся следующие величины. Во-первых, степень нечеткости кластеризации – параметр m . От выбора этого параметра зависит значение функции принадлежности точки кластеру. Обычно он выбирается в пределах $\sim 1.5..2$. Чем больше его значение, тем более «нечеткая» кластеризация. В данном случае эмпирически было выбрано значение $= 1.6$. Во-вторых, выбирается мера расстояний $||x_i - c_j||$. Она характеризует степень близости точки к центру кластера. В нашем случае, так как вектор характеристик состоит только из одного значения и задача, по сути, сводится в выделении значимых интервалов на прямой, то мера расстояния может иметь вид:

$$||x_i - c_j|| = |x_i - c_j| \quad (9)$$

Эта мера характеризует удаленность точки от центра кластера на прямой.

Третий параметр, который выбирается при инициализации – параметр сходимости алгоритма ε (уровень точности). Когда разность значений целевых функций текущей и предыдущей итераций достигнет

этого уровня, считается, что кластеризация завершена. В данном случае, этот параметр был равен 0.001.

Четвертый параметр задается во избежание зависания алгоритма при невозможности достижения уровня точности. Это количество итераций алгоритма. В данном случае этот параметр равен 10000.

После выбора параметров генерируется случайным образом первоначальная матрица принадлежности объектов кластерам и происходит переход ко второму шагу алгоритма.

На шаге 2 происходит вычисление центров кластеров следующим образом:

$$c_j = \frac{\sum_{i=1}^N u_{ij}^{1.6} \cdot x_i}{\sum_{i=1}^N u_{ij}^{1.6}} \quad (10)$$

где c_j - центр j -ого кластера,

N – количество объектов,

x_i - значение i -го объекта,

u_{ij} - степень принадлежности объекта i кластеру j .

На третьем шаге формируется новая матрица принадлежности с учетом вычисленных на предыдущем шаге центров кластеров:

$$u_{ij} = \frac{1}{\sum_{l=1}^c \left(\frac{\|x_i - c_j\|}{\|x_i - c_l\|} \right)^{3.33}} \quad (11)$$

где u_{ij} - степень принадлежности объекта i кластеру j ,

c – количество кластеров

c_j - вектор центра j – го кластера,

c_l - вектор центра l – го кластера.

При этом если для некоторого кластера j и некоторого объекта i , $\|x_i - c_j\| = 0$, тогда полагаем, что степень принадлежности u_{ij} равна 1, а для

всех остальных кластеров степень принадлежности этого объекта равна нулю.

Далее на четвертом шаге вычисляется значение целевой функции, и полученное значение сравнивается со значением на предыдущей итерации. Если разность не превышает заданного в параметрах кластеризации значения ε , считаем, что кластеризация завершена. В противном случае переходим ко второму шагу алгоритма.

Рассмотрим подробнее процесс добавления функции кластерного анализа в нашу программу.

Во-первых, сформируем файлы, на которых будет происходить работа кластеризатора. В один из них запишем список кандидатов в термины:

```
ГРЕК  
РЕКА  
РАК  
РУКА
```

В другой – текст, который будем анализировать:

```
ЕХАТЬ ДРЕВНИЙ ГРЕК ЧЕРЕЗ РЕКА ВИДЕТЬ ДРЕВНИЙ ГРЕК В РЕКА ДРЕВНИЙ  
РАК СУНУТЬ ДРЕВНИЙ ГРЕК РУКА В РЕКА ДРЕВНИЙ РАК ЗА РУКА ДРЕВНИЙ ГРЕК  
ЦАПНУТЬ.
```

Обратите внимание: информация приводится уже в обработанном виде. Слова стоят в едином регистре и в канонических формах.

Первое, что мы делаем – добавляем в меню новый пункт «Кластерный анализ» и записываем в нем код выбора файлов для анализа:

```
try  
{  
    string inpf;  
    OpenFileDialog od = new OpenFileDialog();  
    od.Filter = "Текстовые файлы|*.txt";
```

```

        od.Title = "Выберите файл с текстом для анализа";
        if (od.ShowDialog() == DialogResult.OK)
        {
            inpf = od.FileName;
            od.Title = "Выберите файл со списком кандидатов в термины";
            od.Filter = "Текстовые файлы|*.txt";
            if (od.ShowDialog() == DialogResult.OK)
            {
                SaveFileDialog sv = new SaveFileDialog();
                sv.Filter = "Текстовые файлы|*.txt";
                sv.Title = "Выберите файл для сохранения результата";

                if (sv.ShowDialog() == DialogResult.OK)
                {
                    //код кластерного анализа
                }
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

Теперь на месте комментария «код кластерного анализа» запишем саму реализацию. Допустим, кластеризовать наши термины мы хотим по самому простому параметру – по частоте. Тогда сначала определим для каждого слова частоту его встречи в тексте, записав эту информацию в коллекцию Dictionary. Для этого разобьем считанную из обоих файлов информацию на строковые массивы методом Split, а затем пробежим по массиву слов и с помощью лямбда-выражения опишем запрос, получающий количество встречи текущего слова в массиве-тексте. Для удобства анализируемую информацию запишем в textBox-ы. Код, реализующий это, будет следующим:

```

tbInput.Text = File.ReadAllText(inpfile).ToUpper();
string[] all_text = tbInput.Text.Split(new char[]{' '},
StringSplitOptions.RemoveEmptyEntries);

Dictionary<string, int> temp = new Dictionary<string,
int>();

string[] terms = File.ReadAllLines(od.FileName);
tbBigram.Text = string.Join(" ", terms);

for (int i = 0; i < terms.Length; i++)
{
    if (terms[i].Length > 0)
    {
        var c = all_text.Count(x => x == terms[i]);
        temp.Add(terms[i], c);
    }
}

```

Теперь добавим в наш проект класс, выполняющий кластеризацию. Назовем его *KlasterSystem*. Для добавления класса в проект необходимо сделать следующее. Во-первых, открыть окно *Solution Explorer*. По умолчанию оно располагается справа на вашем экране, но если по каким-то причинам его там нет, можно вызвать его, нажав на кнопку с изображением лупы на верхней панели. Или же найти его в пункте меню *View*. Далее на выделенном жирным имени вашего проекта необходимо щелкнуть правой кнопкой мыши выбрать *Add → Class...* Далее появится новое окно. Внизу, в поле *Name* требуется ввести имя вашего класса.

Для системы кластеризации нам нужны следующие параметры: массив степеней принадлежности, массив характеристик объектов, параметр q в формуле, степень точности, количество итераций, количество кластеров, количество объектов и массив центров кластеров. Тогда код, описывающий поля класса будет следующим:

```

public class KlasterSystem
{
    float[,] StPr;
    int[] objecti;
    float mi; //параметр q
    float ei;
    int Iter;
    int countKlast;
    int countObj;
    float[] CenKlast;
}

```

Так как система кластеризации определена в отдельном файле, то для вывода ей сообщений на форму нам потребуется реализовать специальный механизм: реагирование на события. То есть, когда наша система кластеризации «захочет» что-то вывести на форму, она будет инициировать событие «вывод», на которое будет реагировать наша форма.

В С# классы могут обмениваться между собой сообщениями. Это реализовано через механизм делегатов и событий. Делегат – это класс, а его объект содержит в себе указатели на методы. Механизм событий в С# построен следующим образом. Есть класс, который выполняет какое-либо действие и есть другой класс, который должен при выполнении действия первым классом выполнить свое. Тогда в первом классе определяется событие, а все «заинтересованные» классы на него подписываются. Примером из реальной жизни могут служить оповещения социальной сети «ВКонтакте». Как только у кого-то происходит события, все «заинтересованные» получают об этом уведомления и могут как-то отреагировать.

Объявление делегата очень похоже на объявление методов. Более того, его внешний вид определяет вид методов, которыми классы могут подписаться на события. Синтаксис объявления делегата:

`delegate <тип возвращаемого значения> Имя (<параметры>) ;`

Переменная типа делегат, объявленная в классе – событие. Для их явного отличия от прочих полей используется ключевое слово `event`. Объявленная вне тела класса переменная типа делегат – это просто объект класса делегат.

Когда вы работает с делегатами, вам необходимо выполнить следующие шаги:

1. Определить вид метода, которым заинтересованные классы будут реагировать на событие. Объявить делегат, соответствующий виду методов-реакций.
2. Описать классы, заинтересованные в событии, и описать их методы-реакции.
3. Определить класс, который будет инициировать событие. Создать у него метод, в котором событие будет происходить.
4. Создать поле-событие у основного класса и метод, который будет осуществлять подписку на это событие. Описать вызов события в теле метода основного класса.
5. В основной программе создать объекты требуемых классов, вызвать метод подписки на событие и основной метод, «запускающий» событие.

С программной точки зрения подписка на событие – добавление ссылки на метод к переменной типа делегат, то есть обычное сложение. Синтаксис ее следующий:

`событие+=метод ;`

или

`событие+=переменная-делегат ;`

или

```
событие=new ИмяДелегата(метод) ;
```

Последняя операция выполняется при первой подписке.

Итак, согласно описанному шаблону, определяем вид делегата и описываем его. Выполнить это можно в том же namespace, что содержит описание класса KlasterSystem. Код объявления делегата:

```
public delegate void Vivod(object s);
```

Теперь возвращаемся в форму и описываем у нее метод-реакцию на событие. Пусть мы хотим выводить всю полученную информацию в textBox tbOutPut:

```
public void Vivod(object s)
{
    tbOutPut.Text += s.ToString() + Environment.NewLine;
}
```

Теперь переходим, наконец, к нашей системе кластеризации. Добавляем ей поле-событие:

```
public class KlasterSystem
{
    float[,] StPr;
    int[] objecti;
    float mi;
    float ei;
    int Iter;
    int countKlast;
    int countObj;
    event Vivod ev;
    float[] CenKlast;
}
```

Подписку на событие мы выполним в конструкторе, заодно инициализировав все прочие поля:

```
public KlasterSystem(int ck, int I, float e, float m,Vivod
v,Dictionary<string,int> obj)
{
    countKlast = ck;
    mi = m;
```

```

        ei = e;
        Iter = I;
        CenKlast = new float[countKlast];
        ev = new Vivod(v);
        objecti = obj.Values.ToArray();
        countObj = objecti.Length;
    }

```

Теперь опишем основные методы. Во-первых, нам нужно инициализировать начальные функции принадлежности. Для этого определим у нашей системы вспомогательное поле – генератор случайных чисел:

```

public class KlasterSystem
{
    ...
    float[] CenKlast;
    Random r=new Random();
    ...
}

```

Теперь мы выделим память под матрицу и заполним ее случайными значениями от 0 до 1. Однако так как в результате может быть нарушено ограничение, что сумма степеней принадлежности одного объекта кластерам не должна превышать 1, то опишем метод проверки и корректировки этого:

```

bool Klaster_CheckStPr()
{
    for(int i=0;i<countObj;i++){
        int t=-1;
        for(int j=0;j<countKlast;j++)
        {
            if (StPr[i,j]==1) {t=j;break;}
        }
        if (t>=0){
            for(int j=0;j<countKlast;j++)
            {
                if (j!=t)StPr[i,j]=0;
            }
        }
    }
}

```

```

    }
}

float sum=0;
for(int j=0;j<countKlast;j++)
{
    sum=sum+(StPr[i,j]);
}
while (sum-1>=0.01)
{
    float max=0;
    int ti=0;
    for(int j=0;j<countKlast;j++)
    {
        if (StPr[i,j]>max){max=StPr[i,j];ti=j;}
    }
    float razn=max-sum+1;
    if (razn>=0){StPr[i,ti]=razn;sum=0;}
    if (razn<0){StPr[i,ti]=0;sum=sum-max;}
}
sum=0;
for(int j=0;j<countKlast;j++)
{
    sum=sum+(StPr[i,j]);
}
if (sum>1.01) {ev("Error!!!!"); return false;}
}

return true;
}

```

Теперь же опишем сам метод инициализации начальной матрицы:

```

void InitFirstFunction()
{
    StPr = new float[countObj, countKlast];
    for (int i = 0; i < countObj; i++)
        for (int j = 0; j < countKlast; j++)
            StPr[i,j] = (float)r.NextDouble();
    Klaster_CheckStPr();
}

```


Опишем вспомогательный метод расчета расстояний между двумя точками на прямой характеристик:

```
float Klaster_CountRasst(float x, float c)
{
    return (float) (Math.Abs(x - c));
}
```

Метод расчета оценочной функции:

```
float Klaster_CountOcF(int Znach, int Klast)
{
    float sum = 0f;
    for (int i = 0; i < Znach; i++)
        for (int j = 0; j < Klast; j++)
        {
            m = sum + (float) (Math.Pow(StPr[i, j], mi)
            * Klaster_CountRasst(objecti[i], CenKlast[j]));
        }
    return sum;
}
```

Метод расчета величины U приведен ниже. Обратите внимание, что в нем мы выводим служебное сообщение о значении полученной величины. Это делается для отладки алгоритма и отлова ошибок. На этапе реальной работы этот код можно будет убрать.

```
float Klaster_CountNewU(int t, int j)
{
    float u=0f;

    if (Klaster_CountRasst(objecti[t], CenKlast[j])==0) return 1 ;
else
    {
        for (int i=0;i<countKlast;i++){
            if (Klaster_CountRasst(objecti[t], CenKlast[i])==0)
{return 1;} else

u=u+(float) Math.Pow(Klaster_CountRasst(objecti[t], CenKlast[j])/Klaster
_CountRasst(objecti[t], CenKlast[i]), 2/(mi-1));
```

```

    }
    ev("!!!U in Klaster!!!:" + u.ToString());
    if (u!=0) u=1/u;
    return u;
}
}

```

Метод расчета центров кластеров:

```

float Count_CentKlast(int j)
{
    float c = 0f;
    float c1 = 0f;
    for (int i = 0; i < countObj; i++)
    {
        c = c + (float) (Math.Pow(StPr[i,j], mi)) * objecti[i];
        c1 = c1 + (float) (Math.Pow(StPr[i,j], mi));
    }
    return c / c1;
}

```

Основной метод работы системы кластеризации:

```

public string Work(Dictionary<string,int> objects)
{
    int Shag=0;
    float epx=1000f;
    float OldF=0f;
    float NewF=0f;
    ev("Klaster");
    this.InitFirstFunction();
    ev("InitF");
    for(int j=0;j<this.countKlast;j++)
    {
        this.CenKlast[j]=this.Count_CentKlast(j);
    }
    this.Klaster_CountOcF(this.countObj,this.countKlast);
    while ((Shag<this.Iter)&&(epx>this.ei)) //main cicle
    {
        ev(Shag.ToString());
        Shag++;
    }
}

```

```

        for(int j=0;j<this.countKlast;j++)
    {
        this.CenKlast[j]=this.Count_CentKlast(j);
    }
    for(int ii=0;ii<this.countObj;ii++)
        for (int j=0;j<this.countKlast;j++)
            {
                this.StPr[ii,j]=this.Klaster_CountNewU(ii,j);
            }
        if (!this.Klaster_CheckStPr()) return "Function error";
        NewF=this.Klaster_CountOcF(this.countObj,this.countKlast);
        if (Shag>1) epx=Math.Abs(OldF-NewF);
            OldF=NewF;
        } //main cicle
    ev("Itog");
    for (int i=0;i<this.countObj;i++)
    {
        float maxStPr=this.StPr[i,0];
        int numk=0;
        for(int j=0;j<this.countKlast;j++)
        {
            if(maxStPr<this.StPr[i,j]) {maxStPr=this.StPr[i,j];numk=j;}
        }
        ev("Klaster"+j+": "+getName(objects,objecti[i],j==1)+objecti[i]+"
        "+StPr[i,j]);
        ev("-----");
    }
    }
    return "Finish";
}

```

Обратите внимание на вызов метода getName. Это вспомогательный метод, привязывающий значение термина из словаря к его числовой характеристике. Дело в том, что для кластеризации удобнее работать просто с массивом чисел, нежели чем с Dictionary. Поэтому в нарушение объектно-ориентированного подхода в нашем проекте не был определен отдельный класс «Объект кластеризации», а работа происходит с массивами. Метод getName имеет следующий вид:

```
string getName(Dictionary<string,int> o,int g,bool del)
```

```

{
    string s = o.Where(x => x.Value == g).FirstOrDefault().Key;
    if (del)    o.Remove(s);    return s;
}

```

Теперь в основной программе в меню «Кластерный анализ» мы можем осуществить вызов системы кластеризации и записать в файл результат ее работы:

```

KlasterSystem KlSys = new KlasterSystem(2, 1000, 0.001f, 1.6f,
Vivod,temp);
Vivod(KlSys.Work(temp));
File.WriteAllText(sv.FileName, tbOutPut.Text);

```

Итоговый вид программы представлен на рисунке 10:

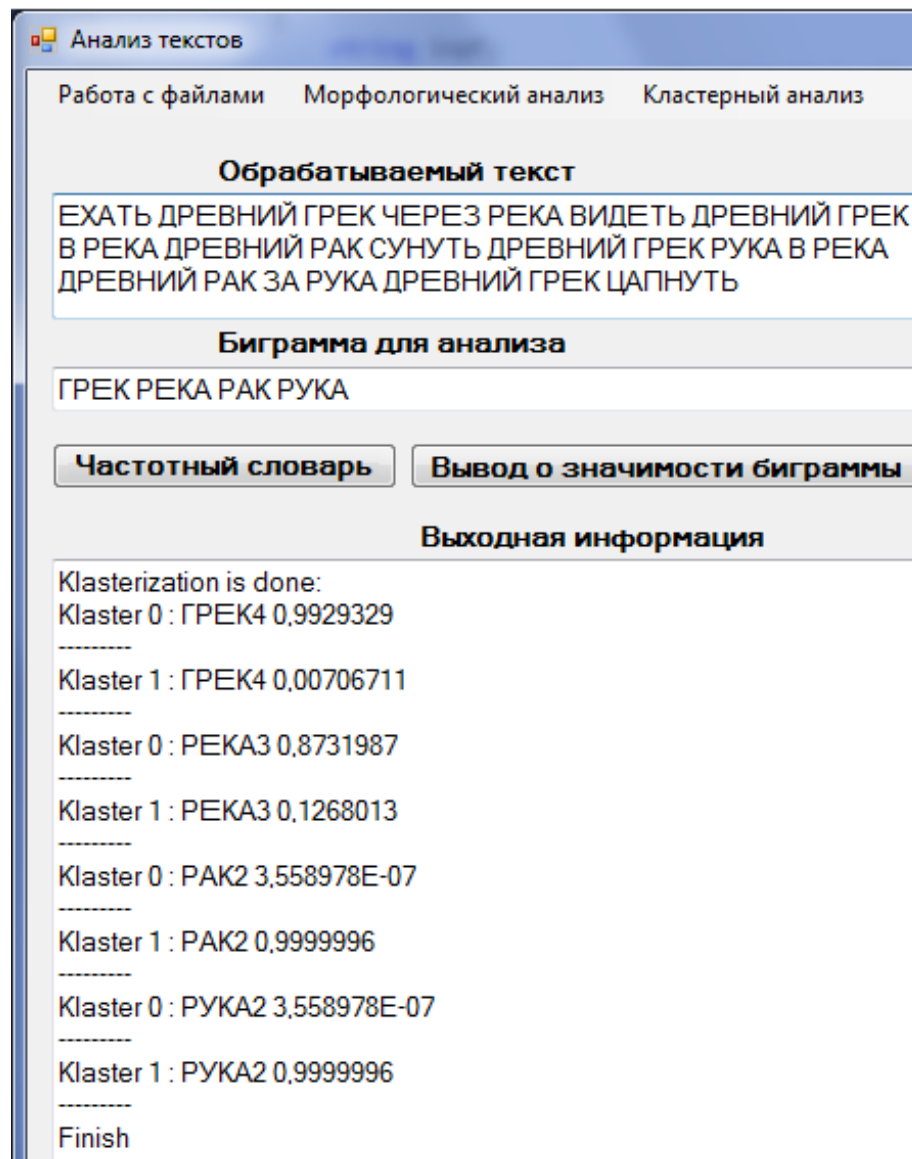


Рис. 10. Результаты кластерного анализа

Мы рассмотрели самый простой вариант кластеризации. Как было сказано выше, FCM-кластеризатор может использовать вычисленные на предыдущих этапах характеристики Frequency, Log-Likelihood, Mutual Information и т.д.

Однако стандартный алгоритм FCM-кластеризации для решения поставленной задачи можно модифицировать. Мы можем взять не одну характеристику, а несколько. Тогда для определения кластера, к которому принадлежит объект, ищется максимум среди средних значений принадлежности объекта к центру каждой характеристики.

Для итогового определения, к какому кластеру принадлежит объект, берется среднее арифметическое значение среди всех характеристик по каждой точке в отдельности. Наибольший результат определяет кластер, к которому была отнесена точка.

$$v_j = \frac{\sum_{i=1}^N u_{ji}}{N}, \quad (12)$$

где u_{ij} - элементы матрицы принадлежности, N - число учитываемых характеристик.

Эту доработку, а также оптимизацию описанного кода вам предлагается выполнить самостоятельно.

Контрольные вопросы к разделу

1. В чем отличие fcm-кластеризации от стандартных алгоритмов?
2. Опишите основные этапы работы алгоритма fcm-кластеризации.
3. Каковы особенности использования механизмов делегатов в C#?

Практические задания к разделу

1. С помощью программы Mystem приведите к канонической форме любой технический текст, содержащий не менее 250 слов. Разбейте текст на двусловия и для каждого из них определите частоту и характеристику MI. Затем проведите кластерный анализ согласно алгоритму fcm-кластеризации по посчитанным величинам.

2. С помощью программы Mystem приведите к канонической форме любой технический текст, содержащий не менее 250 слов. Разбейте текст на двусловия и для каждого из них определите частоту и характеристику T-Score. Затем проведите кластерный анализ согласно алгоритму fcm-кластеризации по посчитанным величинам.

3. С помощью программы Mystem приведите к канонической форме любой технический текст, содержащий не менее 250 слов. Разбейте текст на двусловия и для каждого из них определите частоту и характеристику Log-Likelihood. Затем проведите кластерный анализ согласно алгоритму fcm-кластеризации по посчитанным величинам.

4. С помощью программы Mystem приведите к канонической форме любой технический текст, содержащий не менее 250 слов. Разбейте текст на двусловия и для каждого из них определите частоту и характеристику $TF \times IDF$. Затем проведите кластерный анализ согласно алгоритму fcm-кластеризации по посчитанным величинам.

5. С помощью программы Mystem приведите к канонической форме любой технический текст, содержащий не менее 250 слов. Разбейте текст на двусловия и для каждого из них определите частоту и две любые характеристики. Затем проведите кластерный анализ согласно алгоритму fcm-кластеризации по посчитанным величинам.

ОНТОЛОГИЧЕСКИЙ АНАЛИЗ ТЕКСТОВ

Как было сказано выше, принцип работы существующих алгоритмов извлечения терминологии (term extraction) в лексикографии и терминоведении основан на статистических и лингвистических методах. В основе статистических методов лежит вычисление степени терминологичности на основании числовых закономерностей, присущих термину или нетермину. В основе лингвистических методов лежит отбор по определенным лексико-грамматическим шаблонам и другим лингвистическим признакам термина [7].

Главным недостатком использования статистических и лингвистических методов в процессе извлечения терминологии из текста является отсутствие возможности выделения из получившегося множества терминов только тех, которые относятся к рассматриваемой проблемной области [8].

На множестве информационных единиц в некоторых случаях полезно задавать отношение, характеризующее ситуационную близость информационных единиц, т.е. силу ассоциативной связи между информационными единицами. Его можно было бы назвать отношением релевантности для информационных единиц. Такое отношение дает возможность выделять в информационной базе некоторые типовые ситуации. Отношение релевантности при работе с информационными единицами позволяет находить термины, близкие уже найденным или заранее заданным в общей базе знаний [9].

При анализе больших массивов документации необходимо учитывать специфику её предметной области, чтобы получить в качестве результата список терминов, характерных для конкретной предметной области. Именно для решения подобных задач используют семантические алгоритмы, базирующиеся на определенных семантических метриках.

В настоящее время одной из наиболее универсальных методик представления экспертных знаний с точки зрения полноты

семантического описания информационной единицы предметной области является онтологический подход. Именно поэтому одним из важнейших направлений решения задачи извлечения терминологии из большого массива текстов является разработка и использование семантических метрик на основе онтологических моделей [10].

Рассмотрим описание формальной модели онтологии с примерами из предметной области «Эксплуатация токарно-фрезерного станка с ЧПУ».

Сущность онтологического подхода заключается в том, что предметная область представляется в виде организованной совокупности понятий, их свойств и связей.

Наиболее удобным форматом представления онтологии с точки зрения машинной обработки и наглядности описания особенностей предметной области является язык OWL.

OWL (англ. Web Ontology Language) — язык описания онтологий для семантической паутины. Язык OWL позволяет описывать классы и отношения между ними, присущие веб-документам и приложениям. OWL основан на более ранних языках OIL и DAML+OIL и в настоящее время является рекомендованным консорциумом Всемирной паутины.

В основе языка — представление действительности в модели данных «объект — свойство». OWL пригоден для описания не только веб-страниц, но и любых объектов действительности. Каждому элементу описания в этом языке (в том числе свойствам, связывающим объекты) ставится в соответствие URI. [https://ru.wikipedia.org/wiki/Web_Ontology_Language].

Выделим обязательные требования к OWL - онтологиям, используемой в рамках решения задачи извлечения терминологии:

- Онтология должна наиболее полно отражать особенности объектов предметной области. Задача решается посредством описания максимального количества объектов и классов рассматриваемой области во всем многообразии межклассовых отношений.

- Онтология не должна быть избыточной. Данная проблема решается разбиением отдельного класса на несколько с сохранением семантической целостности за счет определения вспомогательного предиката «имеет Отношение», отражающего некоторое взаимодействие объектов определяемых классов. Примером такого взаимодействия является триплет «Блокировка» - «имеет Отношение» - «Паллета».

- Онтология должна быть наглядной. Это требование в некоторой степени противоречит предыдущему, поэтому поиск компромиссных вариантов представления тех или иных объектов классов онтологии – одна из важнейших задач адекватного отражения особенностей рассматриваемой предметной области [11].

Онтологический подход хранения знаний предполагает представление их в следующем виде:

$$O = \langle T, R, F \rangle \quad (13)$$

где:

T — термины прикладной области, которую описывает онтология. Например, объекты «Резцедержатель», «Станина», «Поплавковое реле».

R — отношения между терминами предметной области, при этом $R \subset \{R_{inc}, R_{add}, R_{term}, R_{lem}, R_{NC}\}$; R_{inc} - множество встроенных отношений объектов таких, как «sameAs» и «SubClassOf»; R_{add} – множество отношений, позволяющих расширять набор объектов описываемой предметной области за счет сочетания лемм связанных объектов. Пример: свойства «имеет Отношение» и «является Частью»; R_{term} - отношение «является Термином», имеющее логический тип значения. Это свойство является вспомогательным и определяется экспертом исходя из критерия - насколько данный объект онтологии является характерным конкретно для этой предметной области. Используется в процессе извлечения терминов согласно тезаурусному критерию терминологичности, R_{lem} - отношение «имеет Лемму», имеющее строковое значение, полученное путем леммирования (приведения к

начальной форме) наименования объекта с помощью программы Mystem компании Яндекс по соответствующим морфологическим признакам термина; R_{NC} – множество отношений объектов, а также свойств типа данных, наиболее полно описывающих особенности взаимодействия объектов рассматриваемой предметной области. Пример: свойства «является Типом Смазки», «является Этапом», «состоит Из».

F — множество функций интерпретации (аксиоматизации), заданных на терминах и/или отношениях онтологии [12]. Примеры таких функций в разработанной онтологии представлены выражениями (14) и (15):

$$F_{COЖ} : X_{ТипСверления} \rightarrow Y_{ТипПодачи}, \quad (14)$$

где $F_{COЖ}$ – отношение «является Типом Поддачи СОЖ», $X_{ТипСверления}$ – множество объектов класса «Тип Сверления», $Y_{ТипПодачи}$ – множество объектов класса «Тип Поддачи СОЖ».

$$F_{InEng} : X_{Контекст} \rightarrow Y_{Eng}, \quad (15)$$

где F_{InEng} – отношение «имеет Английский Эквивалент», $X_{Контекст}$ – множество объектов класса «Контекст», Y_{Eng} – множество объектов класса «Английский Аналог».

Теперь мы можем описать использование семантической метрики оценки терминологичности слов/сочетаний слов. Использование семантической метрики «термин/нетермин» на множестве слов конкретного текста с использованием заранее разработанной OWL-онтологии в процессе извлечения терминологии предполагает определение для каждого поступающего слова или сочетания слов степени близости к терминам рассматриваемой области. Применение такой метрики позволяет выделить из массива поступающих однословий/многословий только те термины и сочетания, которые относятся к данной предметной области.

Степень близости входных слов/сочетаний слов к терминам проблемной области (k_{Ont}) может иметь значение от 0 до 1: чем ближе

полученное значение к 1, тем с большей долей вероятности данное одно-/многословие является термином [13].

В ходе решения поставленной задачи было разработано два критерия выделения терминов из предметной области посредством использования онтологии:

- Тезаурусный критерий;
- Критерий вложенных связей.

Тезаурус представляет собой контролируемый словарь терминов на естественном языке, явно указывающий отношение между терминами и предназначенный для информационного поиска. Любая онтология является усложненной версией тезауруса [14].

Тезаурусный подход к извлечению терминологии предполагает непосредственный поиск вхождений лемм поступающих на вход слов и их сочетаний среди терминов, определенных в онтологии. Для этого в разработанной онтологии для каждого класса определено свойство «имеет Лемму», которое имеет строковое значение, полученное путем леммирования (приведения к начальной форме) имени объекта с помощью программы Mystem компании Яндекс по соответствующим морфологическим признакам термина.

Алгоритм определения степени близости слов/сочетания слов к терминам проблемной области согласно тезаурусному критерию предполагает:

- Оценку степени близости поступающего на вход слова/словосочетания каждому объекту онтологии без учета онтологического критерия оценки.
- Определение опорного объекта онтологии, наиболее близко ассоциирующегося с входным одно-/многословием.

Опорный объект онтологии, используемый в дальнейшем анализе, имеет степень близости по отношению к входному слову/сочетанию слов рассчитанную по следующей формуле:

$$k_t = \max_{i=1}^m \left(\frac{n_i}{p_i} \right), \quad (16)$$

где m – количество всех объектов онтологии;

n_i - число слов из леммы входного многословия, найденных в лемме объекта онтологии;

p_i - общее число слов в лемме объекта онтологии.

Общая схема оценки степени близости слов/сочетания слов терминам проблемной области согласно тезаурусному критерию приведена на рисунке 11.

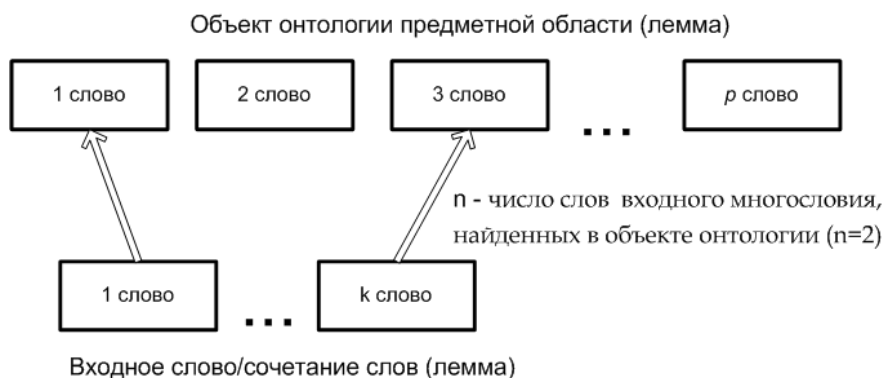


Рис.11. Поиск опорного объекта онтологии

При этом порядок следования слов многословия в опорном объекте должен сохраняться.

Если несколько разных объектов онтологии имеют одинаковое значение коэффициента k_t , то опорным будет считаться тот объект, которому соответствует максимальное n_i . Если таких объектов несколько, то они все будут считаться опорными и анализ по онтологическому критерию будет проведен для каждого из этих объектов.

Структура онтологии рассматриваемой предметной области предполагает наличие у каждого из её объектов свойства (DatatypeProperty) «является Термином», имеющее логический тип значения. Это свойство является вспомогательным и определяется экспертом исходя из критерия - насколько данный объект онтологии является характерным конкретно для этой предметной области.

Степень близости слова/сочетания слов терминам рассматриваемой предметной области в соответствии с тезаурусным критерием оценивается по следующей формуле:

$$k_{Ont} = \frac{k_t}{c + 1}, \quad (17)$$

где k_t – результат первого этапа анализа;

c – число отношений, связывающих опорный объект онтологии с ближайшим объектом, имеющим истинное значение свойства «является Термином».

В случае если сам опорный объект имеет истинное значение данного свойства, то $c=0$. Схема данного поиска приведена на рисунке 12.



Рис.12. Тезаурусный критерий

Таким образом, процесс оценки степени близости одно/многословия к терминам проблемной области по метрике «термин/не термин» в его онтологической составляющей представляет собой движение по графу, в узлах которого находятся объекты соответствующих классов онтологий.

Если опорный объект имеет ложное значение свойства «является Термином» и при этом не имеет никаких связей с другими объектами онтологии, либо все связанные объекты также имеют значение «false»

этого свойства, то находится другой опорный объект для данного слова/словосочетания и оценка проводится заново. В аналогичной ситуации с другими опорными объектами либо в случае их отсутствия входное одно/многословие признается «не термином» ($k_{Ont}=0$).

Критерий вложенных связей. Помимо оценки степени терминологичности отдельно взятого слова/сочетания слов, разработанная метрика позволяет извлечь термины из текста посредством их сопоставления с имеющимися объектами и сочетаниями лемм соответствующих объектов с помощью отношений R_{add} , определенных в онтологии.

Таким образом, при сопоставлении входных сочетаний и объектов предметной области, связанных между собой однонаправленными отношениями R_{add} , термином рассматриваемой предметной области будет считаться многословие, лемма которой полностью совпадает с объединением лемм соответствующих объектов онтологии.

Особенностью данного метода является необходимость представления объектов онтологии преимущественно в виде однословий с максимизацией числа отношений между объектами. Определяющими для использования этого метода являются отношения R_{add} , связь объектов посредством которых позволяет формировать словосочетания естественным образом. Пример формирования многословий с помощью свойства «имеет Отношение»:

1. Найденная цепочка объектов: «Вращение» + «имеет Отношение» + «Двигатель» + «имеет Отношение» + «Переменный ток».
2. Объединение лемм объектов онтологии: «вращение двигатель переменный ток».
3. Термин, извлекаемый из обрабатываемого текста: «вращение двигателя переменного тока».

Пример формирования многословий с помощью свойства «является Частью»:

1. Найденная цепочка объектов: «Подшипник» + «является Частью» + «Шпиндель»;
2. Объединение лемм объектов онтологии: «подшипник шпиндель»;
3. Термин, извлекаемый из обрабатываемого текста: «подшипник шпинделя».

При этом извлеченные термины, входящие в свою очередь в термины, состоящие из большего количества слов, не рассматриваются в качестве терминов с целью избегания избыточности.

Тогда, как следует из описанного выше, для применения семантической метрики анализа текстов необходимо разработать онтологию предметной области. Выполняется это действие экспертом. Инструментом для выполнения действия может быть редактор Protégé 4, являющийся свободно распространяемым, открытым и имеющим легко расширяемую архитектуру за счёт поддержки модулей расширения функциональности. Онтология хранится в файле с расширением .owl.

Пример объявления класса «Концевая фреза» OWL-онтологии для рассматриваемой предметной области:

```
<owl:Class rdf:ID="Концевая_фреза ">
  <rdfs:label
rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >Концевая фреза</rdfs:label>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Фреза"/>
  </rdfs:subClassOf>
</owl:Class>
```

Рассмотрим теперь процесс создания онтологии для рассматриваемой в пособии задачи.

Нам нужны будут следующие сущности: «древний Грека», «река» и «древний рак». Тогда определим классы, объектами которых они будут являться:

1. «древний Грека» – объект класса «Грека», который в свою очередь является подклассом класса «Человек».
2. «река» – объект класса «Река», который в свою очередь является подклассом класса «Водоем».
3. «древний рак» – объект класса «Рак», который в свою очередь является подклассом класса «Животное».

В Protege 4 создание классов осуществляется на вкладке «Classes», показанной на рисунке 13.

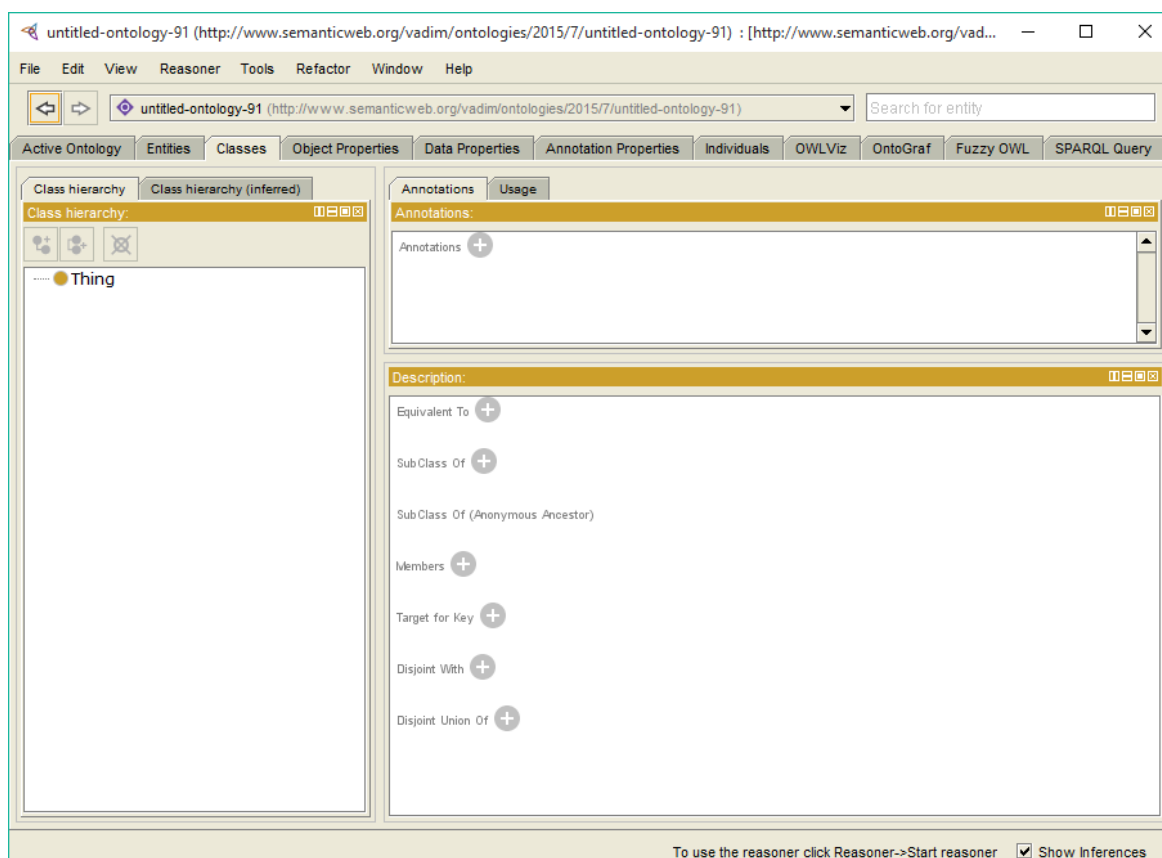


Рис. 13. Вкладка «Classes»

Пустая онтология содержит один класс с именем **THING**. Класс **THING** - это класс, представляющий множество, содержащее все объекты предметной области. По этой причине все классы являются подклассами **THING**.

С помощью кнопок, представленных на рисунке 14, создаем нашу иерархию.



Рис. 14. Создание иерархии классов

При нажатии кнопки создания нового класса появляется диалоговое окно (рисунок 15) для ввода имени нового класса.

Обратите внимание: имена классов, свойств и объектов не должны содержать пробелов и знаков препинания.

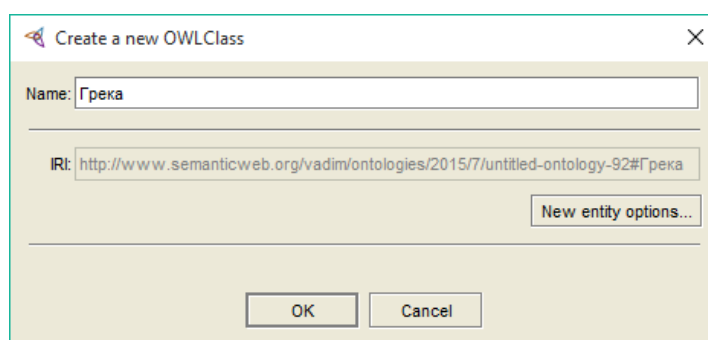


Рис. 15. Окно ввода имени класса

В результате получается следующая иерархия классов – см. рисунок 16.

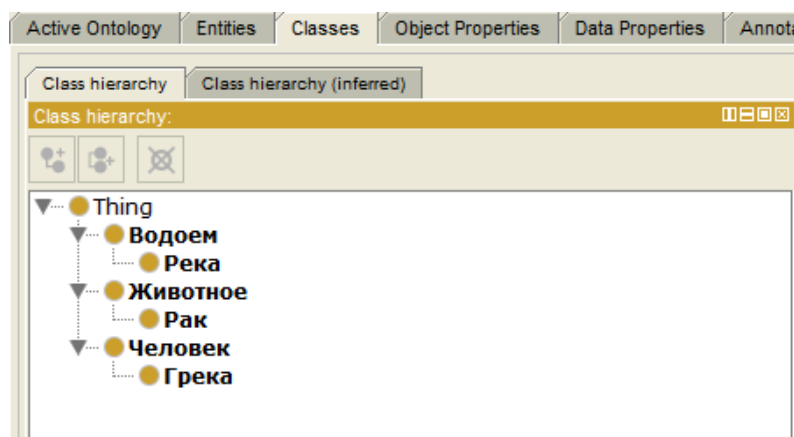


Рис. 16. Полученная иерархия классов

Теперь для каждого класса определим соответствующие свойства.

Существует два основных типа свойств: свойства объектов (Object Properties) и свойства типа данных (Data Type Properties). Свойствами объектов являются отношения между двумя сущностями (индивидами, объектами классов). Свойства Datatype связывают сущность со значением типа данных схемы XML или RDF. Другими словами, они описывают связи между индивидами и значениями данных.

Теперь определим свойства типа данных (Data Type Properties) для созданных классов. Для этого нужно перейти на вкладку Data Properties (см. рисунок 17).

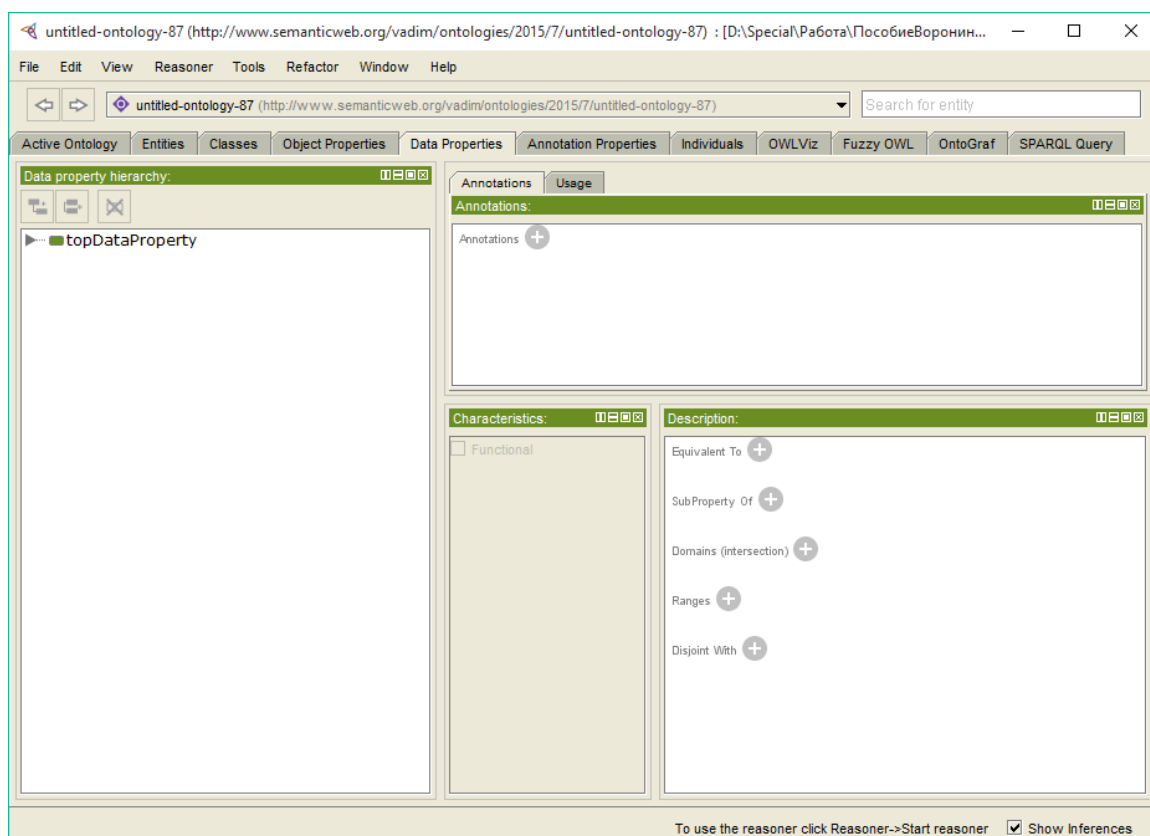



Рис. 17. Вкладка Data Properties

С помощью кнопок  добавляем новые свойства типа данных для наших классов. Первое свойство – «имеетИмя» для созданного класса «Человек». Вводим наименование свойства, как показано на рисунке 18.

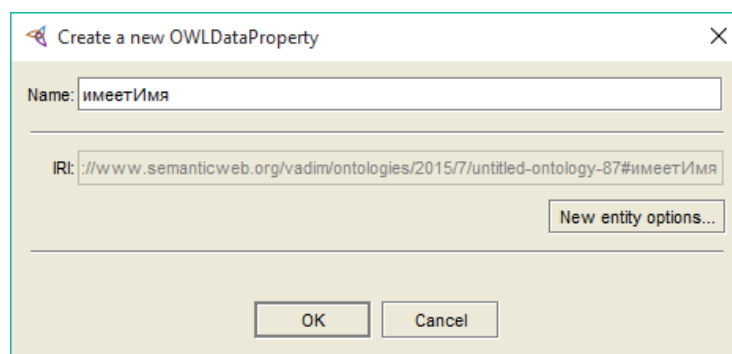


Рис. 18. Создание нового Data Type Property

Свойства могут иметь домен и диапазон. Свойства типа данных определяют тип данных из диапазона для свойства, относящегося к объекту из домена. Т.е. домен имеет значение одного из созданного нами классов, а диапазон – значение встроенного типа данных (string, integer, dateTime, char и т.д.).

Исходя из вышесказанного, определяем для созданного свойства Диапазон («Ranges») - тип «string» и Домен («Domains») – класс «Человек», как показано на рисунке 19.

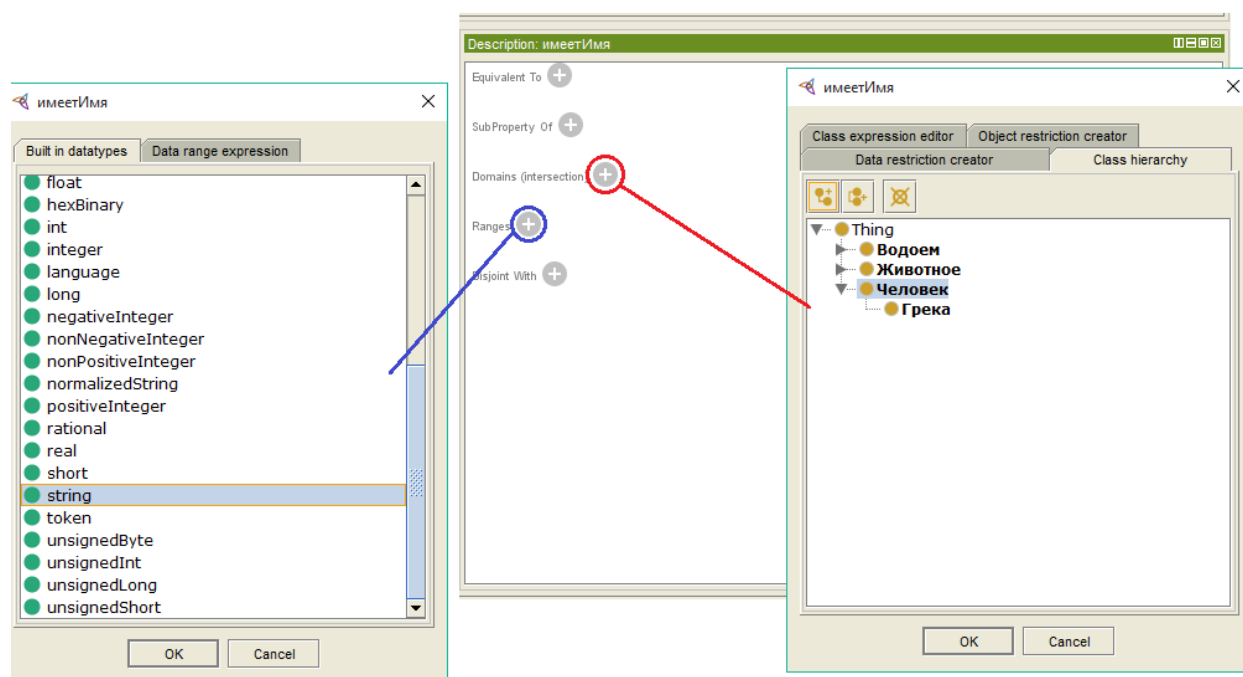


Рис. 19. Определение диапазона и домена для свойства «имеетИмя»

Аналогичным образом создаем остальные свойства типа данных (см. таблицу 8).

Таблица 8. Datatype Properties

Свойство типа данных	Домен	Диапазон
имеетИмя	Класс «Человек»	String
имеетНаименование	Класс «Животное»	String
имеетНазвание	Класс «Водоем»	string

В результате получится следующий набор свойств (см. рисунок 20):

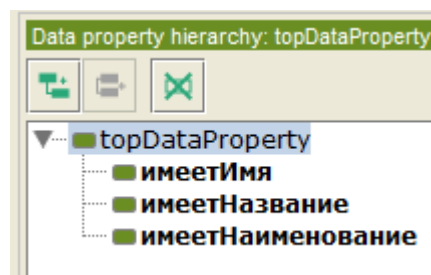


Рис. 20. Свойства типа данных

Теперь определим свойства объектов (Object Properties) для созданных классов. Для этого нужно перейти на вкладку Object Properties (см. рисунок 21).

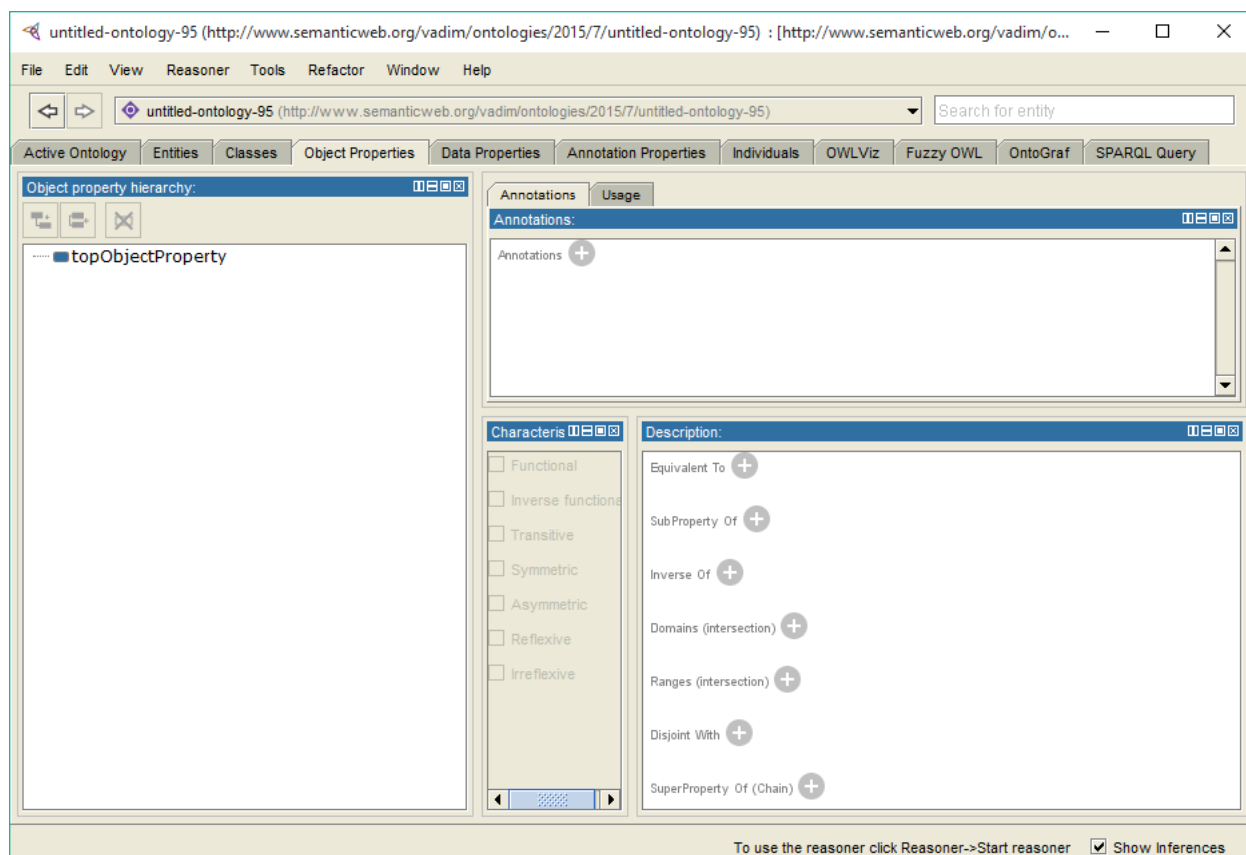


Рис. 21. Вкладка Object Properties


С помощью кнопок  добавляем новые свойства объектов для наших классов. Для решения нашей задачи определим следующие свойства – см. таблицу 9.

Таблица 9. Object Properties

Свойство объектов	Домен	Диапазон
видит	Класс «Человек»	Класс «Животное»
ехалЧерез	Класс «Человек»	Класс «Водоем»
находитсяВ	Класс «Животное»	Класс «Водоем»
сунулРукуВ	Класс «Человек»	Класс «Водоем»
заРукуЦап	Класс «Животное»	Класс «Человек»

В отличие от свойств типа данных, свойства объектов связывают индивидов из доменов с индивидами в диапазонах.

Аналогично свойствам типа данных создаем свойства объектов (см. рисунок 22).

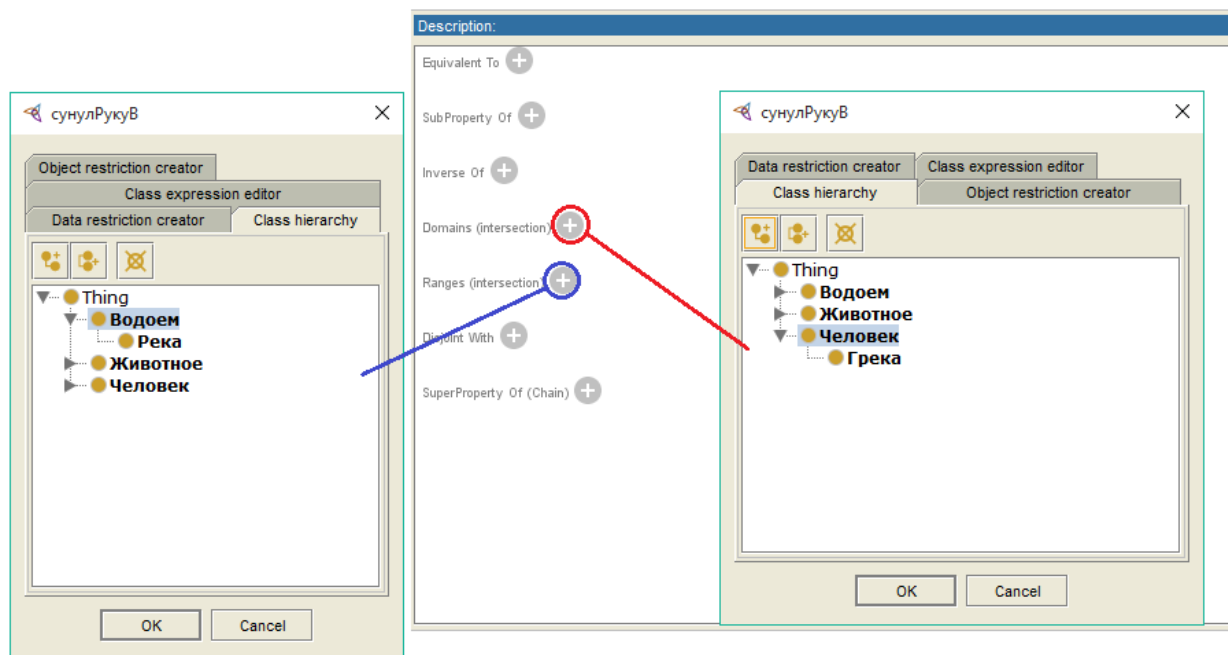


Рис. 22. Определение диапазона и домена для свойства «сунулРукуВ»

В результате получится следующий набор свойств (см. рисунок 23):

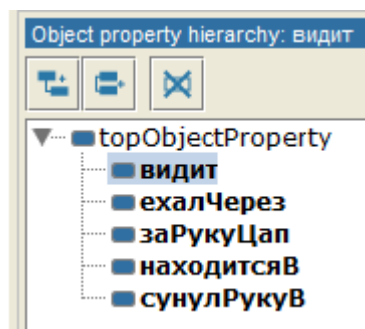


Рис. 23. Свойства объектов

Теперь создадим сами объекты онтологии. Для этого нужно перейти на вкладку Individuals (см. рисунок 24).

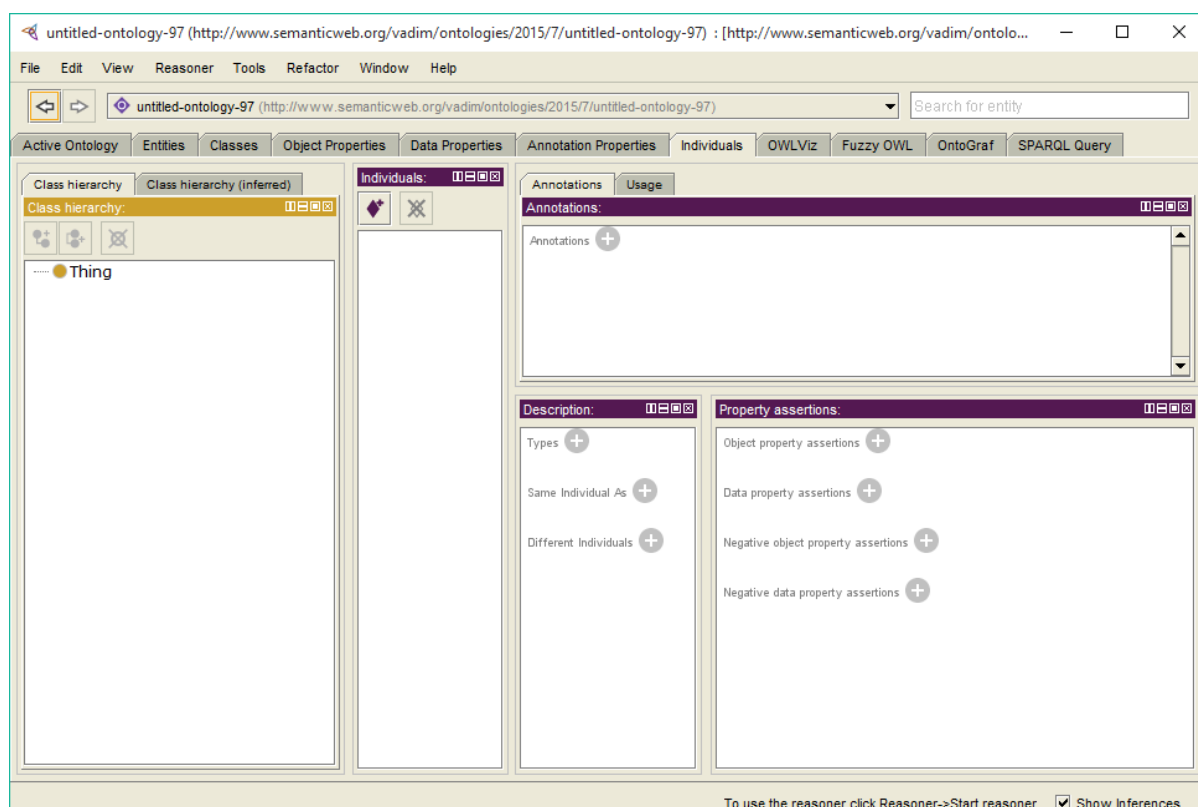



Рис. 24. Вкладка Individuals

Сначала создадим сущность «древний Грека». Для этого на панели Individuals нужно нажать на  и ввести имя объекта в появившемся диалоговом окне (см. рисунок 25).

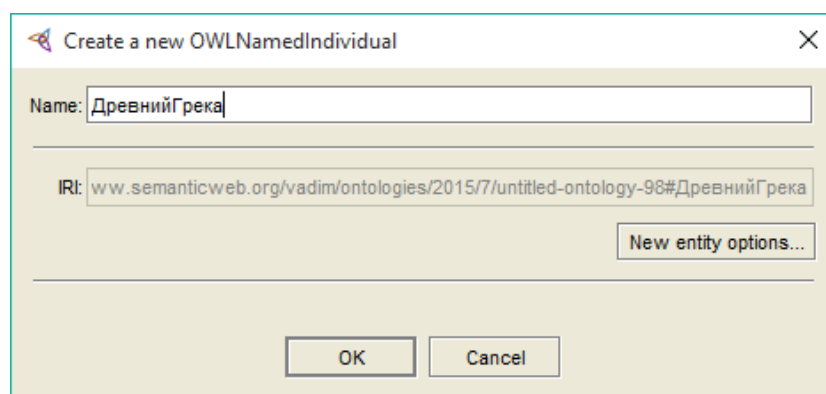



Рис. 25. Создание объекта «ДревнийГрека»

Затем необходимо определить класс, объектом которого является сущность «ДревнийГрека». Для этого на панели Description после нажатия на  у поля Types в диалоговом окне нужно выбрать класс «Грека» (см. рисунок 26).

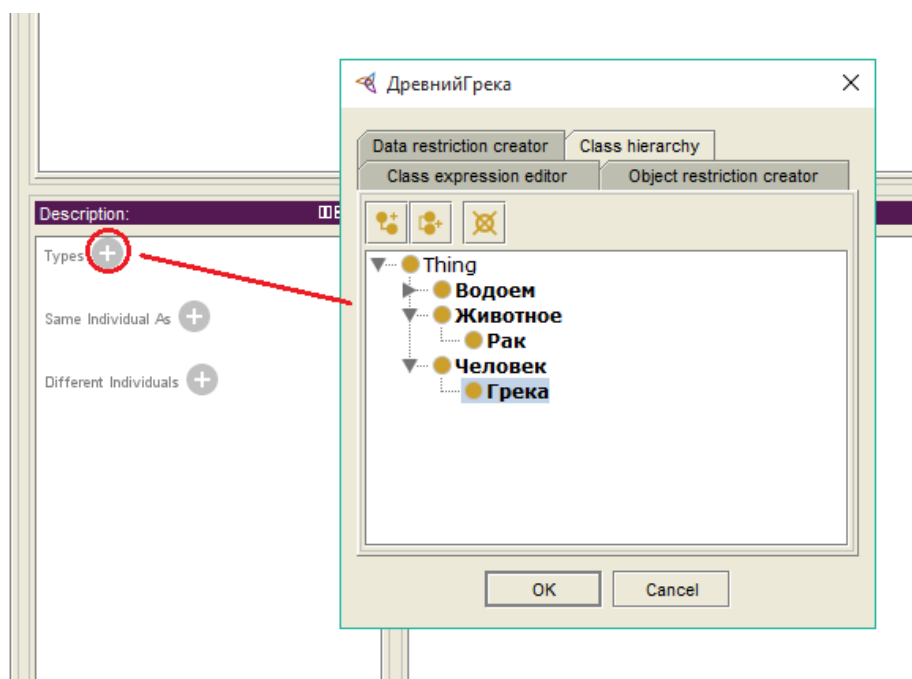



Рис. 26. Определение класс для объекта «ДревнийГрека»

Аналогичным образом создаются сущности «древний рак» (объект класса «Рак») и «река» (объект класса «Река»).

Теперь необходимо задать для каждого из созданных объектов свойства типа данных. Для решения нашей задачи определим следующие свойства (см. таблицу 10).

Таблица 10. Свойства типа данных для объектов онтологии

Объект	Свойство	Значение	Тип данных
ДревнийГрека	имеетИмя	Древний Грека	string
ДревнийРак	имеетНаименование	Древний Рак	string
Река	имеетНазвание	Река	string

Для заполнения свойства «имеетИмя» нужно на панели «Property assertions» нажать на  рядом с полем «Data property assertions». Затем в диалоговом окне на панели «Data Property» выбрать свойство «имеетИмя», выбрать тип данных «String» поля Type и ввести значение «Древний Грека» в поле «Value» (см. рисунок 27). Затем нажать «Ок».

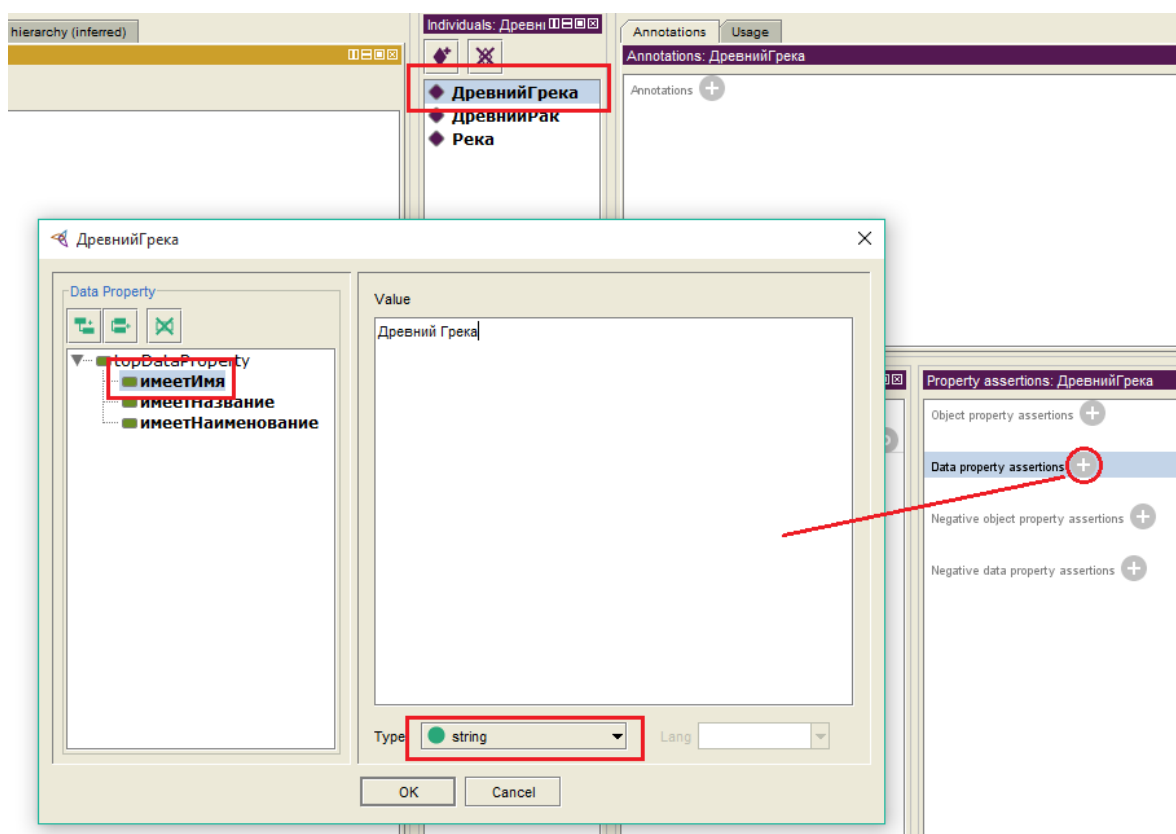



Рис. 27. Инициализация свойства «имеетИмя»

Аналогичным образом задаются и остальные свойства типа данных из таблицы 10.

Теперь необходимо задать отношения между объектами (т.е. свойства объектов). Для решения нашей задачи определим следующие свойства (см. таблицу 11).

Таблица 11. Свойства объектов онтологии

Объект	Свойство	Значение
ДревнийГрека	ехалЧерез	объект «Река»
	видит	объект «ДревнийРак»
ДревнийРак	находитсяВ	объект «Река»
	заРукуЦап	объект «ДревнийГрека»

Для заполнения свойства «ехалЧерез» нужно на панели «Property assertions» нажать на  рядом с полем «Object property assertions». Затем в диалоговом окне в иерархии свойств объектов выбрать свойство «ехалЧерез» и выбрать объект «Река» (см. рисунок 28), затем нажать «Ок».

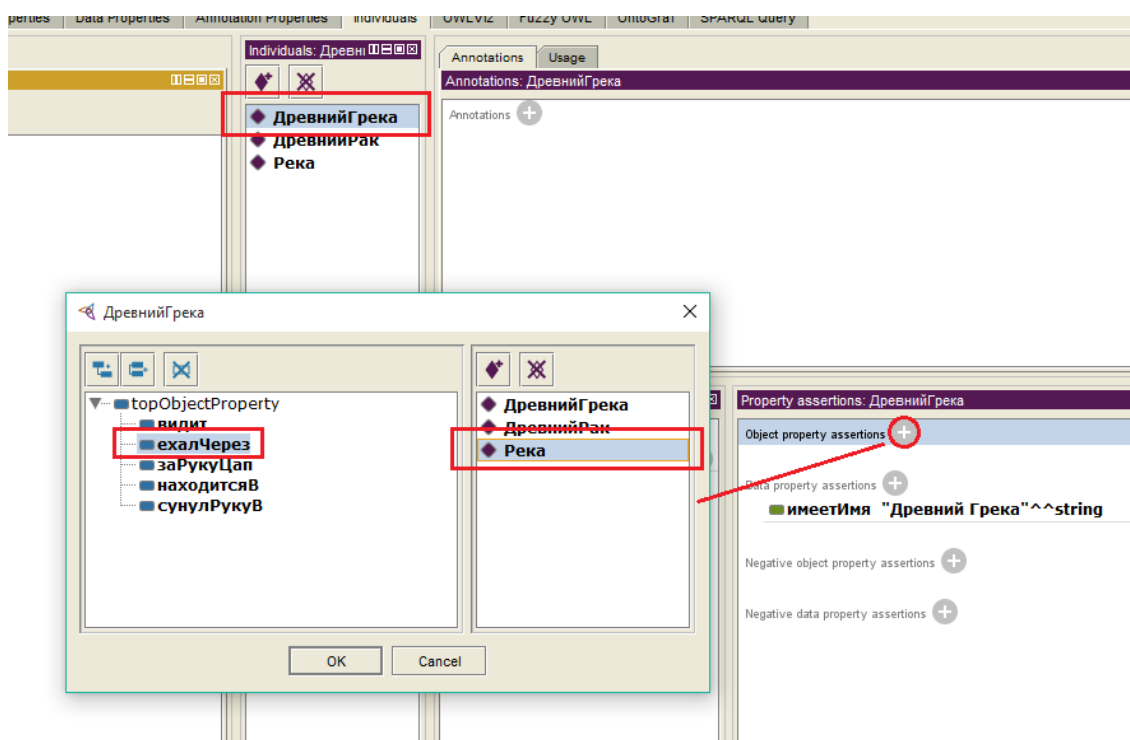


Рис. 28. Инициализация отношения «ехалЧерез»

Аналогичным образом задаются и остальные отношения из табл. 11.

После того, как все необходимые классы, объекты и свойства созданы, можно сохранить разработанную онтологию с помощью File

→ Save as и в появившемся окне выбрать формат OWL/XML (см. рисунок 29).

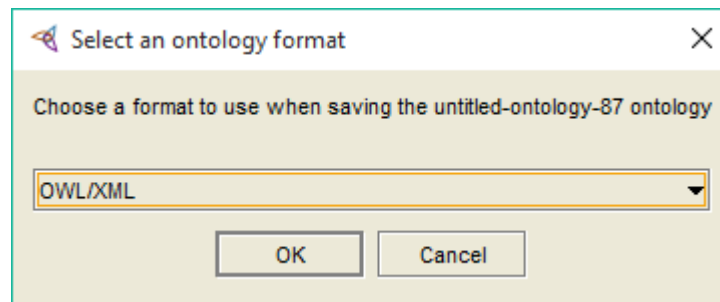


Рис. 29. Сохранение онтологии

В итоге у нас получилась онтология, графическое представление которой изображено на рисунке 30.

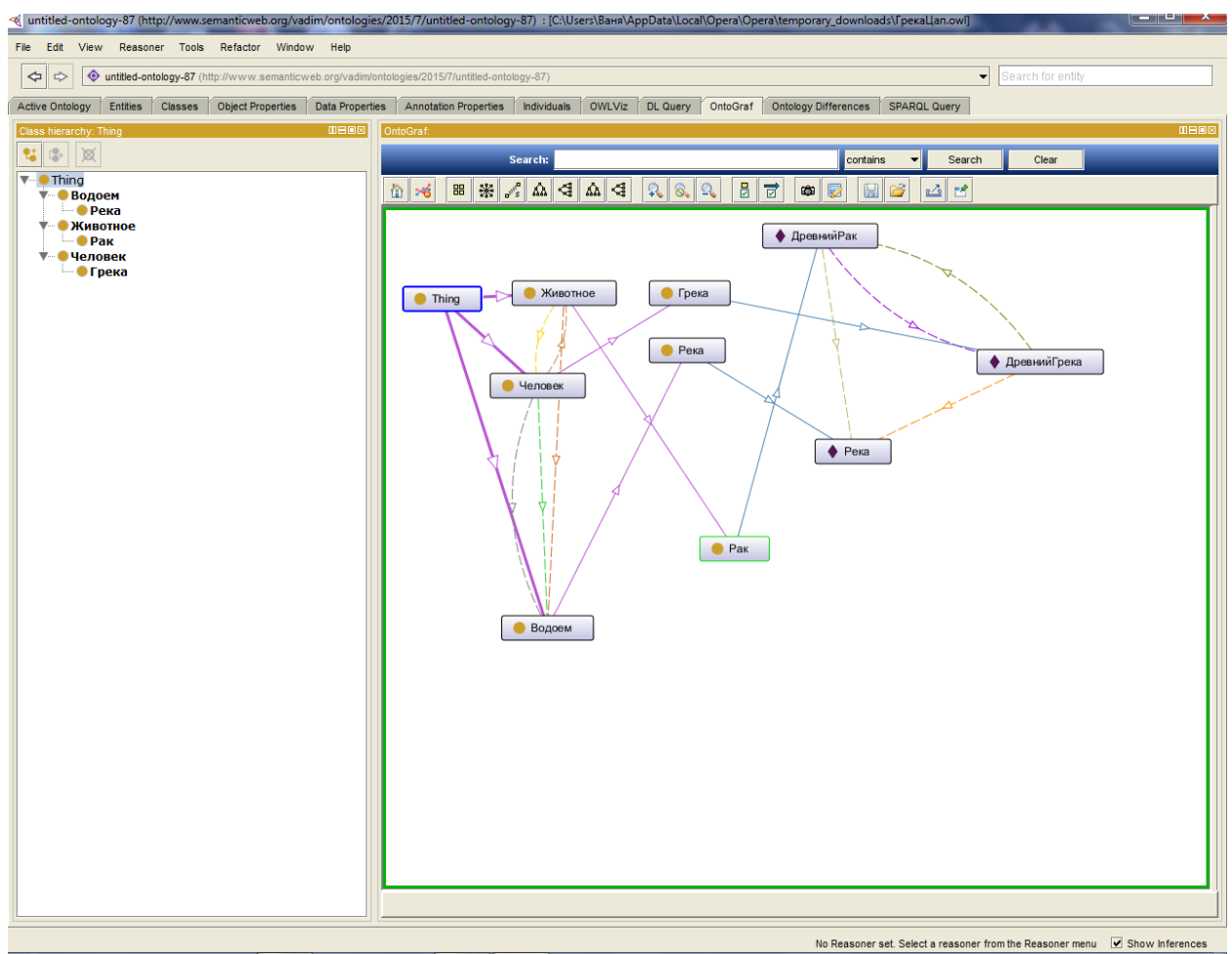


Рис. 30. Графическое представление онтологии

Опишем теперь способы работы с owl-файлами в C#. Так как по сути owl является обычным XML-форматом, то нам подойдут любые классы работы с этими файлами. Но для начала рассмотрим подробнее, что такое XML.

XML (англ. eXtensible Markup Language – расширяемый язык разметки; произносится [икс-эм-эль]). Рекомендован Консорциумом Всемирной паутины (W3C). Спецификация XML описывает XML-документы и частично описывает поведение XML-процессоров (программ, читающих XML-документы и обеспечивающих доступ к их содержимому). XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программам и одновременно удобный для чтения и создания документов человеком, с подчёркиванием нацеленности на использование в Интернете. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями конкретной области, будучи ограниченным лишь синтаксическими правилами языка. Сочетание простого формального синтаксиса, удобства для человека, расширяемости, а также базирование на кодировках Юникод для представления содержания документов привело к широкому использованию как собственно XML, так и множества производных специализированных языков на базе XML в самых разнообразных программных средствах.

Основным понятием в XML будет понятие логической структуры документа, или элемент. Каждый документ содержит один или несколько элементов. Границы элементов представлены начальным и конечным тегами. Имя элемента в начальном и конечном тегах элемента должно совпадать. Элемент может быть также представлен тегом пустого, то есть не включающего в себя другие элементы и символьные данные, элемента. Тег (англ. tag) — конструкция разметки, которая содержит имя элемента. Начальный тег: `<element1>` Конечный тег: `</element1>` Тег пустого элемента: `<empty_element1 />` У элемента могут быть описаны атрибуты. Например: `<element1 attribute="value">` В элементе атрибуты могут использоваться только в начальном теге и теге

пустого элемента. Тэги могут быть вложены друг в друга. Тогда говорят, что охватывающий тэг – родитель вложенных [15].

В C# классы для работы с этим языком разметки описаны в пространстве имен System.Xml. Основная сложность работы с файлами такого формата – необходимость изучения их структуры и написание под нее кода. В приложении приведен код файла созданной нами онтологии для рассматриваемой скороговорки. Давайте рассмотрим его подробнее. Первая часть файла содержит описания используемых элементов: классов, свойств объектов и свойств данных. Они записаны в тэгах Declaration, а сами описываются тэгами Class, ObjectProperty, DataProperty. Само же интересующее нас значение записано в атрибуте IRI. Отношения между классами описаны ниже. Например, в тэге SubClassOf указаны межклассовые отношения. Причем, первый дочерний элемент тэга является подклассом второго. Далее отношения описаны по той же схеме.

Рассмотрим пример кода, извлекающий из файла имена классов. Для этого мы можем использовать объект класса XmlDocument. Добавим в наш проект пункт меню «Онтологический анализ». С помощью OpenFileDialog выберем файл с онтологией и, считав его содержимое в строку, передадим объекту XmlDocument. Затем выгрузим по имени «Class» все элементы документа и выведем в textBox уникальные значения атрибута IRI. Код в данном случае будет следующим:

```
try
{
    OpenFileDialog od = new OpenFileDialog();
    od.Filter = "Файлы-онтологии|*.owl";
    od.Title = "Выберите файл с онтологией";
    if (od.ShowDialog() == DialogResult.OK)
    {
        string ont = File.ReadAllText(od.FileName);
```

```

XmlDocument doc = new XmlDocument();
doc.LoadXml(ont);
XmlNodeList m = doc.GetElementsByTagName("Class");
foreach (XmlNode x in m)
{
    if (!tbOutPut.Text.Contains(x.Attributes["IRI"].Value))
        tbOutPut.Text += x.Attributes["IRI"].Value + Environment.NewLine;
}
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

```

Если же нам нужно выделить межклассовые отношения, то мы можем сделать это следующим образом:

```

XmlNodeList m = doc.GetElementsByTagName("SubClassOf");
foreach (XmlNode x in m)
{
    tbOutPut.Text += x.FirstChild.Attributes["IRI"].Value
+ Environment.NewLine;
    tbOutPut.Text += x.LastChild.Attributes["IRI"].Value
+ Environment.NewLine;
}

```

Теперь рассмотрим подробнее работу алгоритмов онтологического анализа нашей скороговорки. Начнем с тезаурусного критерия. Обратите внимание, что при использовании наших алгоритмов для каждого объекта онтологии нужно создать Datatype Property «имеетЛемму», в котором будут записываться начальные формы названий объектов. Значения отношения «имеетЛемму» - «Древний Грек», «Река», «Древний Рак». Кроме того, для тезаурусного критерия предположим, что все объекты являются терминами (отношение «являетсяТермином»=истина). Тогда, если мы имеем на входе, например, «Старый грека», то лемма = «Старый грек». Алгоритм будет работать следующим образом. Он находит среди объектов онтологии по

значению отношения «имеетЛемму» опорный, т.е. тот, который содержит наибольшее количество слов из входного сочетания. Для нашего примера – «Древний грек». Далее по формуле 16 происходит расчет выхода, который для данного примера будет равен 0,5.

Расшифруем подробнее расчет по приведенной формуле. $n_i=1$, количество слов-совпадений – «Грек», $p_i = 2$ – количество слов в найденном объекте «Древний грек». Тогда в итоге получаем 0,5. Программную реализацию данного примера вам предлагается осуществить самостоятельно.

Теперь рассмотрим работу второго алгоритма – критерия вложенных связей, который всегда выдает 0 или 1. Пусть входным значением будет «Древний Грека видит Древнего Рака в реке». Лемма входного значения: «Древний Грек видеть Древний рак река». Тогда алгоритм находит объект «Древний Грек» – полное совпадение лемм, и смотрит, есть ли отношения между этим объектом и объектом «Древний Рак». В нашем случае он находит его. Пусть у нас уже найдено «Древний Грек»+связь+«Древний Рак». Тогда смотрим, есть ли связь между «Древний Рак» и «Река». По нашей онтологии получаем: есть связь – «находится». В результате найдено «Древний Грек»+связь+«Древний Рак»+связь+ «Река». Сочетание полностью найдено, вывод – 1. Если хотя бы одно звено цепи не найдено, вывод – 0. Программную реализацию данного примера вам также предлагается осуществить самостоятельно.

Для проверки правильности программной реализации алгоритмов приведем контрольные примеры для каждого из критериев.

Тезаурусный критерий:

- 1.Вход: «Старый грек», выход: 0,5.
- 2.Вход: «Течет река», выход: 1.
- 3.Вход: «Рак сильно постарел», выход: 0,5.

Критерий вложенных связей:

1. Вход: «Древний Грека видит Древнего Рака в реке», выход: 1.

2.Вход: «Рак за руку цап грека», выход: 0.

3.Вход: «Древний Рак за руку цап древнего грека», выход: 1.

Контрольные вопросы к разделу

1. Какие требования должна удовлетворять онтология при решении задачи извлечения терминологии?
2. Какие типы свойств существуют в owl-онтологиях?
3. Может ли одна сущность онтологии быть объектом нескольких классов?

Практические задания к разделу

1. С помощью редактора Protégé разработайте OWL-онтологию учебного процесса университета. Онтология должна включать в себя такие классы как «Преподаватель», «Студент», «Кафедра», «Дисциплина», «Расписание занятий» и т.д.
2. Программно реализуйте алгоритм тезаурусного критерия для owl-онтологии «Грека и Рак». Определите степень терминологичности следующих словосочетаний: «старый грек», «течет река», «рак сильно постарел»;
3. Программно реализуйте алгоритм критерия вложенных связей для owl-онтологии «Грека и Рак». Определите степень терминологичности следующих словосочетаний: «Древний Грека видит Древнего рака в реке», «Рак за руку цап грека», «Древний Рак за руку цап древнего грека».

РЕАЛИЗАЦИЯ ДОПОЛНИТЕЛЬНЫХ ВОЗМОЖНОСТЕЙ

Вывод отчетов в формате PDF

Любое приложение, занимающееся анализом данных, должно представлять отчеты о своей работе. Причем, желательно наглядные и в удобном для пользователя формате. Одним из способов реализации этой возможности является использование формата PDF.

Аббревиатура PDF расшифровывается как Portable Document Format и представляет собой кроссплатформенный формат электронных документов, созданный фирмой Adobe Systems с использованием ряда возможностей языка PostScript. Формат предназначен, в первую очередь, для представления в электронном виде полиграфической продукции. Просмотр документов, созданных в этом формате, осуществляется через специальную программу Adobe Reader, а также программы сторонних разработчиков, которые являются официально бесплатными. С 1 июля 2008 года формат PDF является открытым стандартом ISO 32000.

Функциональные возможности формата достаточно богаты. Он позволяет внедрять в документ необходимые шрифты, изображения, формы, а также мультимедиа-вставки. Имеет собственные технические форматы для полиграфии и позволяет использовать механизмы электронных подписей для защиты и проверки подлинности документов.

Так как любые внедренные объекты могут существенно увеличить объем документа, то рекомендуется использовать векторную графику и «безопасные» шрифты. Всего имеется 14 таких шрифтов. Это шрифты семейства Times: обычный, курсив, полужирный и полужирный курсив; семейства Courier: обычный, наклонный, полужирный и полужирный наклонный; семейства Helvetica: обычный, наклонный, полужирный и полужирный наклонный; а также Symbol и Zapf Dingbats. Эти шрифты будут корректно отображаться любыми программами без

необходимости их внедрения. Если используемый в документе шрифт не был внедрен и отсутствует в системе, то это может повлечь ошибки отображения.

В первое время своего существования данный формат был крайне непопулярен, так как, во-первых, программное обеспечение от Adobe для чтения и создания его было платным. Во-вторых, в нем отсутствовала поддержка внешних ссылок, что делало его практически бесполезным в сети Интернет. В-третьих, создаваемые документы были большего размера по сравнению с обычным текстом, что означало более длительную загрузку и отображение с заметными задержками. После выпуска бесплатного программного обеспечения популярность формата стала заметно возрастать, и в настоящее время появилось множество задач, связанных с обработкой этого формата. Это создание и изменение pdf-документов, печать, заполнение полей форм и т. д.

В данном разделе нам предстоит рассмотреть функциональные возможности одной из наиболее популярных библиотек для работы с форматом pdf на языке C#. Речь пойдет об iTextSharp, позволяющей создавать pdf-документы и манипулировать ими. Эта бесплатная библиотека является наиболее упоминаемой в Интернет-сообществе [16]. Ее функционал позволяет осуществлять передачу pdf в браузер, генерировать динамические документы из xml-файлов и баз данных, манипулировать страницами pdf-документа, автоматизировать заполнение форм и добавлять цифровую подпись в документ. Единственная задача, реализация которой в данной библиотеке практически не поддерживается, – извлечение текста из pdf-документа.

Рассмотрим пример формирования отчета о частотном анализе текста, а так же об анализе значимости биграммы. Начнем с создания документа. Первое, что необходимо сделать, подключить к проекту библиотеку itextsharp.dll. Подключение библиотеки к проекту осуществляется нажатием правой кнопки мыши на элементе Reference в окне SolutionExplorer (см.рисунок 31).

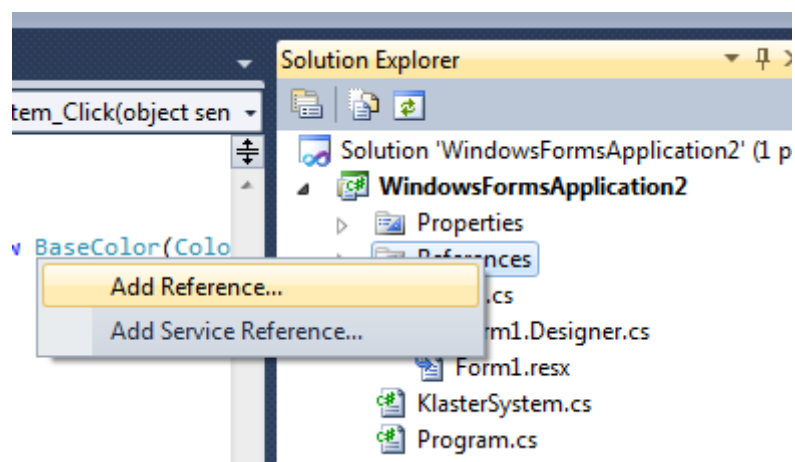


Рис. 31. Подключение библиотеки

Затем в появившемся окне (рисунок 32) на вкладке Browse выбираем путь к библиотеке. Ее рекомендуется заранее положить в папку с исполняемым файлом проекта.

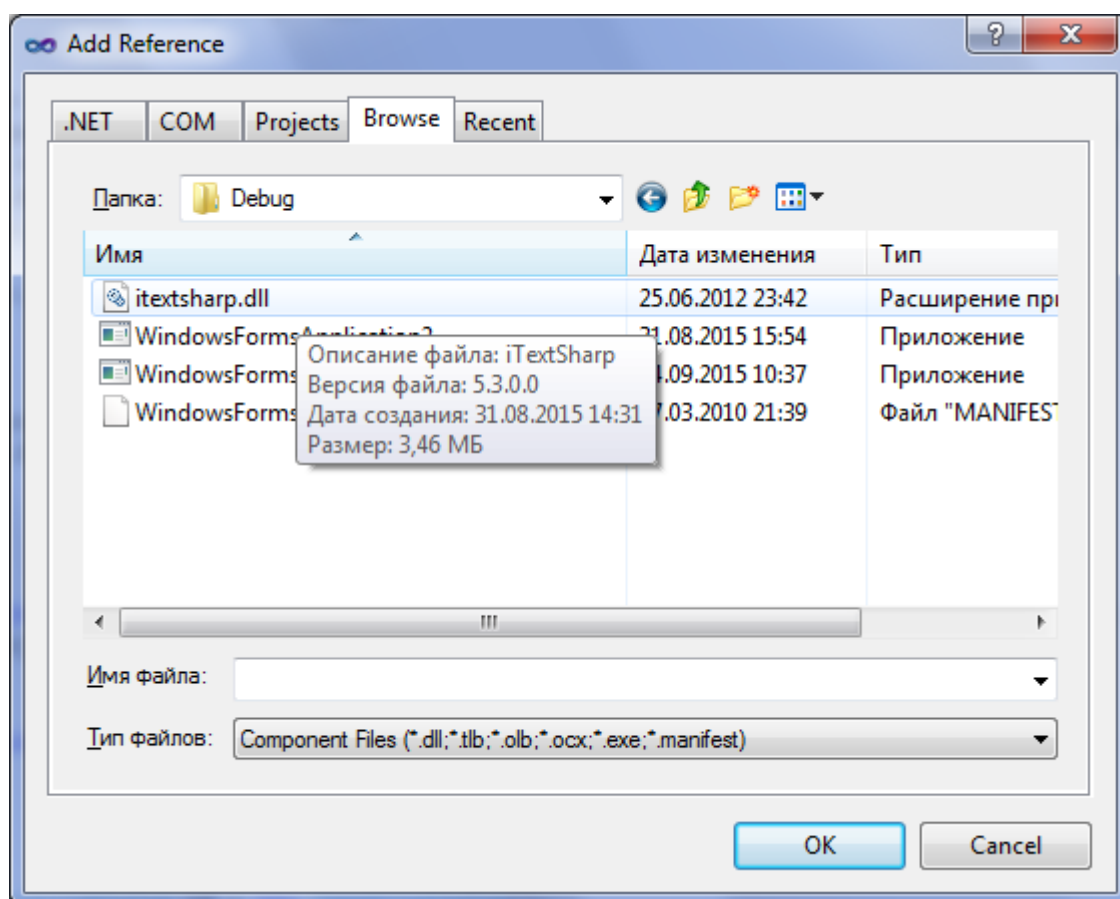


Рис.32. Выбор библиотеки

После того, как библиотека подключена, мы можем прописать ссылки на следующие пространства имен:

```
using iTextSharp.text;
using iTextSharp;
```

```
using System.IO;
using iTextSharp.text.pdf;
```

Теперь необходимо обеспечить работу с документом на логическом и физическом уровне. В данном случае физический уровень представляет собой файловый поток записи на диск, а логический уровень – связанный с потоком объект, позволяющий манипулировать содержимым документа. Реализуем это следующим образом:

```
var doc = new Document();
PdfWriter.GetInstance(doc,
new FileStream(Application.StartupPath+@"\Document.pdf",
FileMode.Create));
doc.Open();
```

Документ в pdf-формате в самом простом варианте может содержать главы, абзацы, таблицы и рисунки. Мы рассмотрим работу с тремя последними объектами. Рассмотрим именно создание заголовка отчета через объект «параграф». Для начала нам необходимо создать объект типа «фраза», который представляет собой текст параграфа. Затем создаем сам параграф и помещаем его в документ. Помещение любого объекта в документ осуществляется вызовом метода Add. При работе с текстом у вас могут возникнуть проблемы с кириллицей. Для их решения необходимо подгрузить в ваш документ файл, содержащий шрифт. Для этого создается объект класса BaseFont, на основе файла *.TTF, который также рекомендуется положить в папку с исполняемым файлом проекта. Тогда код создания заголовка отчета будет следующим:

```
BaseFont baseFont = BaseFont.CreateFont(Application.StartupPath +
@"\ARIAL.TTF", BaseFont.IDENTITY_H, BaseFont.NOT_EMBEDDED);
iTextSharp.text.Phrase j = new Phrase("Результирующий отчет",
new iTextSharp.text.Font(baseFont, 15,
iTextSharp.text.Font.BOLDITALIC, new BaseColor(Color.Green)));
Paragraph a1 = new Paragraph(j);
a1.Alignment = 1;
doc.Add(a1);
```

При добавлении других параграфов в документ объект BaseFont можно уже не создавать. Например, вывод анализируемого текста можно осуществить с помощью следующего кода:

```
j = new Phrase("Текст для анализа:",
               new iTextSharp.text.Font(baseFont, 12,
               iTextSharp.text.Font.BOLDITALIC, new BaseColor(Color.Red)));
a1 = new Paragraph(j);
a1.Alignment = 0;
doc.Add(a1);

j = new Phrase(tbInput.Text,
               new iTextSharp.text.Font(baseFont, 11,
               iTextSharp.text.Font.BOLDITALIC, new BaseColor(Color.Black)));
a1 = new Paragraph(j);
a1.Alignment = 0;
doc.Add(a1);
```

Для вывода графика в отчет нам необходим объект класса Image. Он создается на основе файла в формате jpg. Для получения этого файла, содержащего наш график, мы вызываем метод Save у картинки pictureBox. Код помещения графика в отчет будет следующим:

```
j = new Phrase("Графическое отображение словаря:" +
Environment.NewLine,
               new iTextSharp.text.Font(baseFont, 12,
               iTextSharp.text.Font.BOLDITALIC, new BaseColor(Color.Red)));
a1 = new Paragraph(j);
doc.Add(a1);

pictureBox1.Image.Save(Application.StartupPath + @"\1.jpg");
iTextSharp.text.Image jmg =
iTextSharp.text.Image.GetInstance(Application.StartupPath
+ @"\1.jpg");
jmg.Alignment = Element.ALIGN_CENTER;
doc.Add(jmg);
```

Теперь рассмотрим вывод частотного словаря через работу с таблицами, представленными объектом класса PdfPTable. Конструктор в качестве параметров принимает количество столбцов в будущей таблице, дальше идет последовательное построчное добавление ячеек, представленных объектами класса PdfPCell. Ячейка может быть создана на основе фразы или картинки. Добавление ее в таблицу осуществляется вызовом метода AddCell. Причем этот метод может в качестве параметра принимать простую строку, тогда на ее основе автоматически создастся и добавится ячейка с параметрами по умолчанию. Для ячеек, созданных на основе фразы, параметры (например, шрифт) можно задавать самостоятельно. Из ключевых свойств объекта ячейки необходимо выделить BackgroundColor, Padding, HorizontalAlignment, VerticalAlignment, Colspan и Rowspan. Первое отвечает за цвет заливки ячейки. Второе – за отступы содержимого от границ, третье и четвертое – за вертикальное и горизонтальное выравнивание соответственно. С помощью пятого и шестого свойств происходит объединение ячеек, также как в таблицах на языке HTML. Colspan отвечает за то, на сколько ячеек должна расшириться данная по горизонтали, а Rowspan – по вертикали. Например, для объединения трех столбцов в один необходимо поставить свойство Colspan у первой ячейки в значение 3. Ниже приведен пример добавления таблицы частотного словаря в документ:

```
j = new Phrase("Частотный словарь:",  
new iTextSharp.text.Font(baseFont, 12,  
iTextSharp.text.Font.BOLDITALIC, new BaseColor(Color.Red)));  
a1 = new Paragraph(j);  
a1.Alignment = 0;  
doc.Add(a1);  
  
doc.Add(new Paragraph(" "));  
PdfPTable table = new PdfPTable(2);  
PdfPCell cell = new PdfPCell(new Phrase("Слово",  
new iTextSharp.text.Font(baseFont, 12,
```

```

iTextSharp.text.Font.NORMAL, new BaseColor(Color.Black))));
        cell.BackgroundColor = new BaseColor(Color.Yellow);
        cell.Padding = 5;
        cell.HorizontalAlignment = Element.ALIGN_CENTER;
        table.AddCell(cell);

        cell = new PdfPCell(new Phrase("Частота",
        new iTextSharp.text.Font(baseFont, 12,
iTextSharp.text.Font.NORMAL, new BaseColor(Color.Black))));
        cell.BackgroundColor = new BaseColor(Color.Yellow);
        cell.Padding = 5;
        cell.HorizontalAlignment = Element.ALIGN_CENTER;
        table.AddCell(cell);

foreach (KeyValuePair<string, int> i in CurrentVac)
    {
cell = new PdfPCell(new Phrase(i.Key,
        new iTextSharp.text.Font(baseFont, 11,
iTextSharp.text.Font.NORMAL, new BaseColor(Color.Black))));
        cell.Padding = 5;
        cell.HorizontalAlignment = Element.ALIGN_CENTER;
        table.AddCell(cell);

        cell = new PdfPCell(new
Phrase(i.Value.ToString(),
        new iTextSharp.text.Font(baseFont, 11,
iTextSharp.text.Font.NORMAL, new BaseColor(Color.Black))));

        cell.Padding = 5;
        cell.HorizontalAlignment = Element.ALIGN_CENTER;
        table.AddCell(cell);
    }

```

И как всегда, не забудьте, что таблицу нужно добавить в документ методом Add.

```
doc.Add(table);
```

После того как вы заполнили содержимое документа, необходимо сохранить его. Выполняет это следующая строчка кода:

```
doc.Close();
```

Если в вашей операционной системе задано приложение по умолчанию для открытия файлов в формате pdf, то вы можете открыть отчет сразу после его формирования. Выполняется это следующей строкой кода:

```
System.Diagnostics.Process.Start(Application.StartupPath +  
@"\Document.pdf");
```

Результат работы программы представлен на рисунке 33.

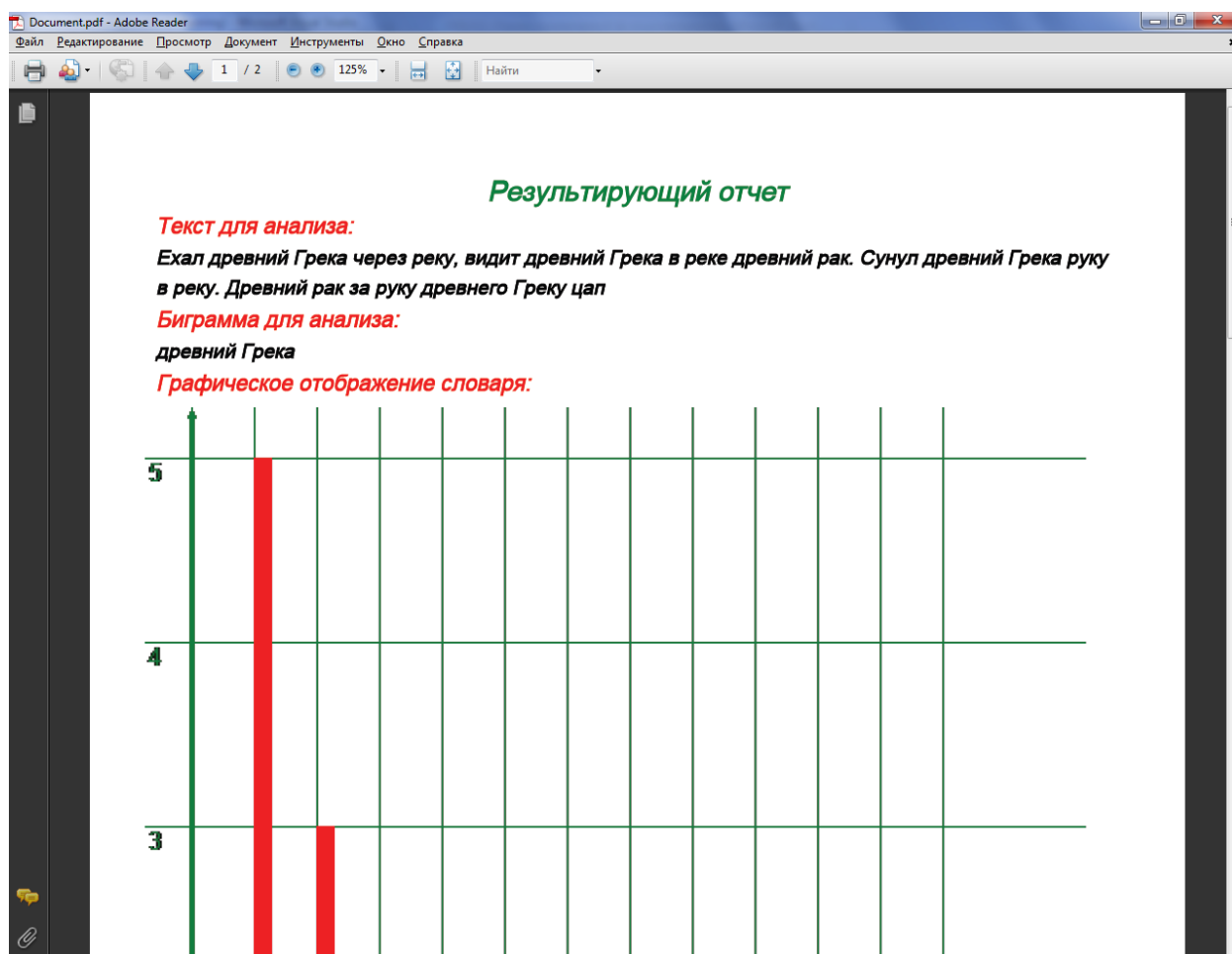


Рис.33. Отчет в формате pdf

Обратите внимание, когда вы подключите библиотеку работы с pdf, у вас может возникнуть следующая ошибка в коде построения графиков:

```
«Error 1 'Font' is an ambiguous reference between  
'System.Drawing.Font' and 'iTextSharp.text.Font'»
```

Суть ошибки: класс Font определен и в пространстве имен System.Drawing и в пространстве имен iTextSharp.text. Для устранения ошибки один из классов необходимо именовать полным именем, то есть добавить к его имени имя пространства имен. Например, сделаем это для кода вывода графиков, тогда он примет следующий вид:

```
gr.DrawString((y).ToString(), new System.Drawing.Font("Arial",  
10), Brushes.Green, 0, (int)(max - (y) * cdy));
```

Мы рассмотрели процесс формирования отчетов о работе программы. Теперь рассмотрим считывание информации для анализа из различных форматов.

Работа с текстами для анализа в формате Word

В настоящее время большинство документов на предприятиях создаются с помощью редактора Microsoft Word. Microsoft Word (часто – MS Word, WinWord или просто Word) – текстовый процессор, предназначенный для создания, просмотра и редактирования текстовых документов, с локальным применением простейших форм таблично-матричных алгоритмов. Выпускается корпорацией Microsoft в составе пакета Microsoft Office. Первая версия была написана Ричардом Броди (Richard Brodie) для IBM PC, использующих DOS, в 1983 году. Позднее выпускались версии для Apple Macintosh (1984), SCO UNIX и Microsoft Windows (1989). Текущей версией является Microsoft Office Word 2013 для Windows и Microsoft Office Word 2011 для Mac [17].

Мы рассмотрим вариант работы со стандартными библиотеками, поставляемыми в составе Visual Studio [18]. При их использовании мы добавляем в проект ссылку на Microsoft.Office.Interop.Word, находящуюся обычно на вкладке .Net (см. рисунок 34). Затем для упрощения работы мы вводим алиас пространства имен Word:

```
using Word = Microsoft.Office.Interop.Word;
```

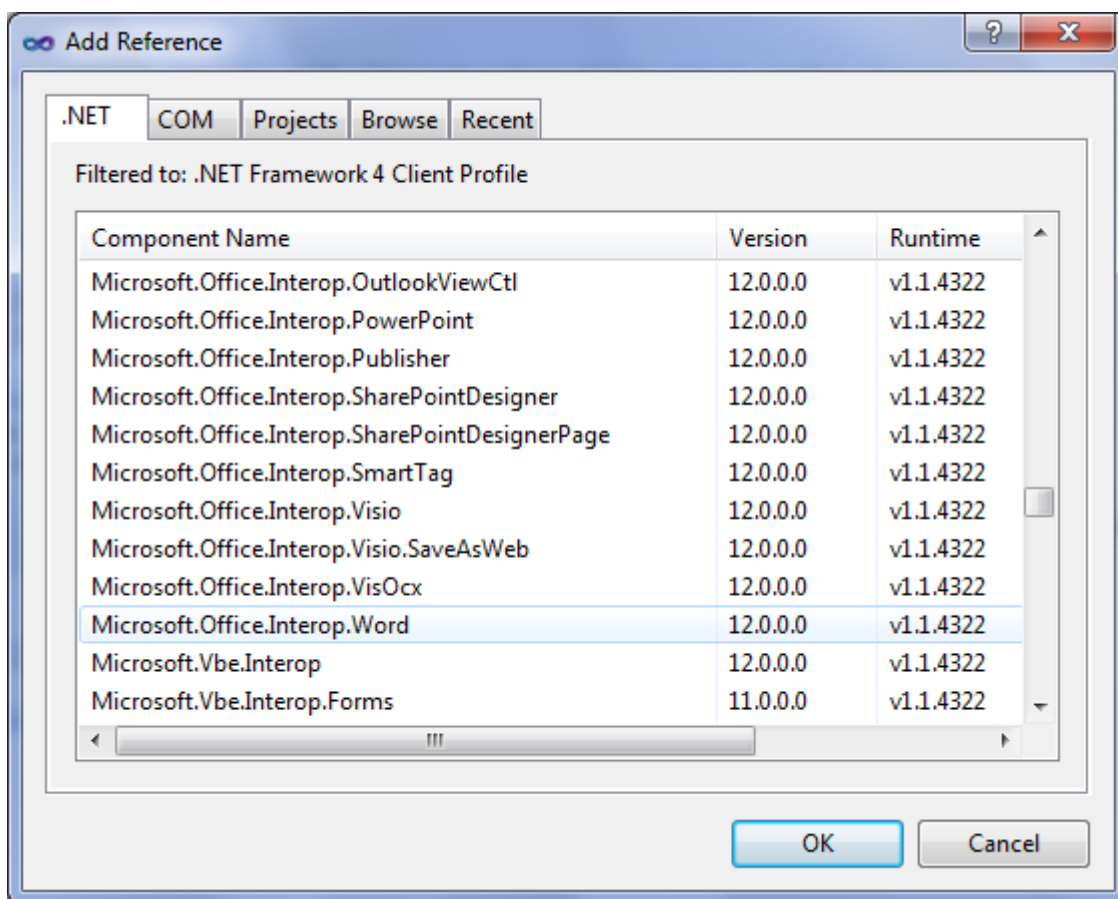



Рис. 34. Добавление ссылки на библиотеку Word

Объекты у Word иерархически упорядочены. Основной элемент: объект Application – это COM-сервер и оболочка для других объектов. Он может содержать коллекцию под названием Documents, хранящую ссылки на объекты типа Document, каждый объект типа Document будет хранить коллекцию Paragraphs или ссылок на объекты типа Paragraph, Table, Range, Bookmark, Chapter, Word, Sentence и т. д. Работа с объектами осуществляется путем использования их свойств и методов.

Запуск Word:

```
Word.Application wordapp = new Word.Application();
wordapp.Visible = true;
```

Закрытие осуществляется методом Quit, имеющим следующие параметры:

- SaveChanges – определяет, как сохраняет Word измененные документы перед осуществлением выхода. Может быть одна из Word.WdSaveOptions констант: wdDoNotSaveChanges – не сохранять

документ, wdPromptToSaveChanges – выдать запрос перед сохранением, wdSaveChanges – сохранить без предупреждения.

- OriginalFormat – необязательный параметр, определяет формат сохранения для документа. Возможна одна из следующих констант: Word.WdOriginalFormat констант: wdOriginalDocumentFormat – в оригинальном формате документа (не изменяя его), wdPromptUser – по выбору пользователя (актуально при открытии документа, а не при его создании, при создании и сохранении окно сохранения документа всегда присутствует при отсутствии заданного имени документа), wdWordDocument – формат .doc.

- RouteDocument – необязательный параметр. При true документ направляется следующему получателю, если документ является attached документом.

Пример кода:

```
Object saveChanges = Word.WdSaveOptions.wdPromptToSaveChanges;  
Object originalFormat = Word.WdOriginalFormat.wdWordDocument;  
Object routeDocument = Type.Missing;  
wordapp.Quit(ref saveChanges, ref originalFormat, ref  
routeDocument);  
wordapp = null;
```

Для открытия существующего документа основным методом является метод Open. Рассмотрим его параметры:

- ref FileName – полный путь к открываемому файлу.
- ref ConfirmConversions – при значении true в случае открытия документа не формата Word будет выводиться диалоговое окно конвертирования файла.
- ref ReadOnly – при значении true документ открывается только для чтения.
- ref AddToRecentFiles – при значении true имя открываемого файла добавляется в список недавно открытых файлов в меню «Файл».
- ref PasswordDocument – пароль открываемого документа.
- ref PasswordTemplate – пароль шаблона документа.

- `ref Revert` – при значении `true` возможно повторное открытие экземпляра того же документа с потерей изменений в открытом ранее. При значении `false` новый экземпляр не открывается.

- `ref WritePasswordDocument` – пароль для сохранения документа.

- `ref WritePasswordTemplate` – пароль для сохранения шаблона.

- `ref Format` – формат открытия. Одна из следующих констант типа `Word.WdOpenFormat`: `wdOpenFormatAllWord`, `wdOpenFormatAuto`, `wdOpenFormatDocument`, `wdOpenFormatEncodedText`, `wdOpenFormatRTF`, `wdOpenFormatTemplate`, `wdOpenFormatText`, `wdOpenFormatUnicodeText` или `wdOpenFormatWebPages`. По умолчанию имеет значение `wdOpenFormatAuto`.

- `ref Encoding` – кодовая страница, или набор символов, для просмотра документа.

- `ref Visible` – при значении `true` документ открывается как видимый.

- `ref OpenAndRepair` – при значении `true` делается попытка восстановить поврежденный документ.

- `ref DocumentDirection` – направление текста. Одна из констант `Word.WdDocumentDirection`: `WdLeftToRight`, `WdRightToLeft`.

- `ref NoEncodingDialog` – при значении `true` подавляется показ диалогового окна `Encoding`, которое отображается, если кодировка не распознана.

- `ref xmlTransform` – определяет тип XML-данных при проводимых XML-преобразованиях.

Для работы с текстовой информацией нам необходимо обратиться к параграфам документа `Word`. Один объект `Paragraph` представляет один абзац текста. Для манипуляции с параграфами нам понадобятся объекты:

```
Word.Paragraphs wordparagraphs;
```

```
Word.Paragraph wordparagraph;
```

Любой документ Word всегда имеет хотя бы один параграф. Даже если он пуст, то всегда присутствует курсор ввода. Вывод текста выполняется не просто в параграф, а в диапазон параграфа – объект Range. Текстовая информация хранится в его свойстве Text.

Рассмотрим пример кода, реализующего выгрузку текста для анализа из doc-файла. Алгоритм его работы предельно прост: с помощью OpenFileDialog мы выбираем doc-файл, затем создаем объект WordApplication, открываем документ и, пробегая по коллекции его параграфов, выгружаем текст каждого в textBox. Реализуется это следующим образом:

```
try
{
    OpenFileDialog od = new OpenFileDialog();
    od.Title = "Выберите файл";
    if (od.ShowDialog() == DialogResult.OK)
    {
        Word.Application wordapp = new Word.Application();
        wordapp.Visible = true;
        wordapp.Documents.Open(od.FileName);
        for(int i=1;i< wordapp.Documents[1].Paragraphs.Count;i++)
            tbOutPut.Text +=
            wordapp.Documents[1].Paragraphs[i].Range.Text.ToString()
            +Environment.NewLine;
        wordapp.Quit();
    }
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Рассмотренный вариант является одним из возможных. Однако далеко не всегда информация хранится в формате Word. В настоящее время на рынке существует множество конкурирующих продуктов.

Одним из таких является OpenOffice. Поэтому дальше рассмотрим работу и с его форматом хранения текстовых документов.

Работа с текстами для анализа в формате OpenOffice

В настоящее время возрастает популярность бесплатного программного обеспечения. Во многих офисах вместо платных программ устанавливают бесплатные аналоги. Одним из таких является свободный пакет офисных приложений Apache OpenOffice. В настоящее время он успешно конкурирует как на уровне форматов, так и на уровне интерфейса пользователя с коммерческими офисными пакетами. В состав OpenOffice входят следующие модули:

- Writer – это текстовый процессор и визуальный редактор HTML, успешно конкурирующий с Microsoft Word, Pages, AbiWord, KWord;
- Calc – табличный процессор, схожий с Microsoft Excel, Numbers, Gnumeric, KSpread;
- Impress – модуль для подготовки презентаций, похожий на Microsoft PowerPoint, Keynote, KPresenter;
- Base – модуль работы с БД, в чем-то даже превосходящий Microsoft Access и Kexi;
- Draw – векторный редактор, способный конкурировать с Microsoft Visio, Microsoft Expression, Desigh, Adobe Illustrator, CorelDRAW, Kivio, Dia;
- Math – редактор формул, схожий по функционалу с MathType, Microsoft Equation Tools, KFormula.

Существует два способа взаимодействия приложения .Net и OpenOffice: использование библиотек (CLI-сборок), поставляющихся с пакетом или с использованием технологии отражения. Первый способ, несомненно, удобнее в использовании для разработчика, но имеет ряд ограничений. Во-первых, он не будет работать с ранними версиями пакета, во-вторых, версия библиотек может конфликтовать с версией Framework, в-третьих, версия библиотек должна совпадать с версией

OpenOffice, т. е. возможна проблема в переносимости приложения с одной машины на другую. Второй способ не обладает описанными недостатками, но существенно более сложен в использовании, так как необходимо знать все методы и свойства внутреннего языка пакета и их параметры. Рассмотрим оба метода более подробно и начнем с использования CLI-сборок.

Первое, что нам необходимо сделать, – подключить данные сборки к нашему проекту. Для этого выполняем следующие действия. Во-первых, находим эти библиотеки в каталоге, где установлен OpenOffice, в папке с именем «assembly». Там должны находиться следующие файлы:

- cli_basetypes.dll
- cli_cppuhelper.dll
- cli_types.dll
- cli_ure.dll

Скопируйте их и переместитесь в каталог проекта. Здесь создайте папку с именем «Resources», в которую и переместите сборки. Теперь подключите эти библиотеки к своему проекту (в обозревателе решений Solution Explorer щелкните правой кнопкой мыши на пункте «References», выберите «Add references...» и найдите свои сборки).

Для более удобной работы с классами из сборок вам необходимо подключить следующие пространства имен:

```
using unoidl.com.sun.star.lang;  
using unoidl.com.sun.star.uno;  
using unoidl.com.sun.star.bridge;  
using unoidl.com.sun.star.frame;
```

Основным объектом для подключений к OpenOffice.org является ServiceManager. Он служит точкой входа для приложения и передается каждому компоненту при создании экземпляра. Часто компоненту требуется больше информации или функциональности после развертывания. В этом ключе возможности ServiceManager ограничены.

Поэтому была создана концепция `ComponentContext`. Он представляет собой хранилище именованных объектов с доступом только для чтения.

Один из таких объектов – `ServiceManager`. То есть по сути это среда, в которой живут компоненты приложения (как область памяти для переменных программы). `ComponentContext` передается приложению при его запуске. Поэтому первое, что нам необходимо сделать для работы с OpenOffice, это запустить его. Делается это вызовом статического метода `bootstrap` класса `Bootstrap`, возвращающего ссылку на объект `ComponentContext`:

```
unoidl.com.sun.star.uno.XComponentContext localContext =  
    uno.util.Bootstrap.bootstrap();
```

Затем мы должны получить ссылку на объект диспетчера служб `ServiceManager` (экземпляр класса `MultiServiceFactory`), для того чтобы позже получить объект класса `Desktop` и создать новый объект `CLI`:

```
unoidl.com.sun.star.lang.XMultiServiceFactory multiServiceFactory  
= (unoidl.com.sun.star.lang.XMultiServiceFactory)  
    localContext.getServiceManager();
```

`Desktop` – центральный управляющий экземпляр для платформы приложений OpenOffice.org. Все их окна организованы как иерархия фреймов, содержащих видимые компоненты. `Desktop` – корень этой иерархии. Через этот компонент можно загрузить видимые компоненты, завершить их работу или перенаправить запросы. Для создания нового экземпляра `Desktop` мы используем наш диспетчер служб, передав команду загрузчику `XComponent`:

```
XComponentLoader componentLoader = (XComponentLoader)  
multiServiceFactory.createInstance("com.sun.star.frame.Desktop");
```

Итак, как видно из сказанного выше, работу OpenOffice обеспечивают три объекта: компонент, диспетчер служб и `Desktop`.

Теперь перейдем к открытию документа `writer`, используя наш объект `Desktop`:

```
XComponent xComponent = componentLoader.loadComponentFromURL(  
    @"file:///"+od.FileName.Replace('\\','/'), "_blank",
```

```
0, new unoidl.com.sun.star.beans.PropertyValue[0] );
```

Рассмотрим подробнее параметры этого метода. Первый содержит адрес того, что нужно загрузить. Второй параметр отвечает за то, в каком окне открыть загружаемый компонент: в новом или в одном из ранее открытых. Третий и четвертый – дополнительные параметры открытия.

Теперь рассмотрим чтение текста из документа модуля writer. Основным элементом в данном случае является текстовое поле. Соответственно, сначала обращаемся к нему, вызывая метод `getText`, затем для чтения текста используем метод `getString`.

```
string s=  
( (unoidl.com.sun.star.text.XTextDocument) xComponent ) .getText()  
.getString();
```

Обратите внимание на то, что ваше приложение, возможно, будет работать медленно при первом запуске OpenOffice. Для того чтобы его ускорить, рекомендуется самостоятельно перед началом работы открыть OpenOffice, дать ему пару минут поработать и затем закрыть. Это происходит из-за того, что при первом запуске происходит поиск обновлений в Интернет.

Мы рассмотрели работу с OpenOffice с использованием библиотек [19], теперь перейдем к рассмотрению второго метода: использование отражения [20].

Сначала рассмотрим теоретические аспекты данной технологии. Механизм отражения позволяет получать объекты типа `Type`, описывающие сборки, модули и типы. Эту технологию можно использовать для того, чтобы динамически создать экземпляр типа, привязать тип к существующему объекту, получить тип из существующего объекта и вызвать его методы или получить доступ к его полям и свойствам. При наличии атрибутов отражение обеспечивает доступ и к ним. При работе с OpenOffice данный способ не столь удобен, как вариант с CLI, но свои плюсы в нем также есть. Для

использования технологии отражения необходимо подключить только namespace System.Reflection.

Механизм работы с приложением в данном случае следующий: создается COM-объект (сом-сервер) OpenOffice в оперативной памяти компьютера. Ваша программа посылает ему команды на выполнение методов или установку/чтение каких-либо свойств объектов, находящихся внутри сервера. Осуществляется это вызовом у объекта, хранящего ссылку на ваш сервер, метода InvokeMember со следующими параметрами:

- строка-название метода или свойства, понятного серверу;
- флаг типа интерпретации первого параметра: строка содержит имя метода или свойства;
- пустая ссылка для работы с OpenOffice (null), а в общем случае может содержать ссылку на дополнительный объект-посредник;
- ссылка на сам сом-сервер;
- массив элементов типа object, хранящий параметры метода или значение свойства;

Теперь рассмотрим практический пример. Первым шагом подключаем пространство имен:

```
using System.Reflection;
```

Теперь получим ссылку на ServiceManager, как сом-сервер OpenOffice. Для этого совершаем следующие действия. Во-первых, выясняем его тип по идентификатору, а затем загружаем экземпляр в оперативную память:

```
Type tServiceManager =  
Type.GetTypeFromProgID("com.sun.star.ServiceManager", true);  
object oServiceManager =  
    System.Activator.CreateInstance(tServiceManager);
```

Для простоты дальнейшего объяснения заключим вызов метода `InvokeMember`, описанного выше, в оболочку метода `InvokeObj`, который будет иметь следующий вид:

```
public object InvokeObj(object obj, string method, BindingFlags binding, params object[] par)
{
    return obj.GetType().InvokeMember(method, binding, null, obj, par);
}
```

Далее для работы с «Writer» нам нужен объект `Desktop`. Получим ссылку на него, вызвав через технологию отражения соответствующий метод у объекта `ServiceMeneger`:

```
object oDesktop = InvokeObj(oServiceManager,
                             "createinstance",
                             BindingFlags.InvokeMethod,
                             "com.sun.star.frame.Desktop");
```

Теперь загрузим документ `writer`. Как мы помним из описанного выше, для этого нам необходим метод `loadComponentFromUrl` с определенными параметрами. Первым шагом объявим массив для их хранения и заполним его:

```
Object[] arg = new Object[4];
arg[1] = "_blank";
arg[0] = @"file:///"+od.FileName.Replace('\\', '/');
arg[2] = 0;
arg[3] = new Object[] { };
```

Теперь вызовем метод `InvokeObj` у `Desktop` с соответствующими параметрами:

```
object oComponent =
InvokeObj(oDesktop, "loadComponentFromUrl",
BindingFlags.InvokeMethod, arg);
```

Теперь обратимся к тексту данного документа через вызов метода `getText` у загруженного компонента. Обратите внимание, что, даже если мы не передаем методу ни одного параметра, массив с параметрами мы все-таки должны объявить, пусть даже пустой:

```
arg = new Object[0];
Object oText =
```

```
InvokeObj(oComponent, "getText",
BindingFlags.InvokeMethod, arg);
```

Для считывания текста используем код:

```
oText = InvokeObj(oText,
"getString",
BindingFlags.InvokeMethod,
arg
);
tbInput.Text = oText.ToString();
```

Закрывать OpenOffice после работы можно следующим образом:

```
InvokeObj(oComponent, "close", BindingFlags.InvokeMethod, new
object[1] { "true"});
```

Здесь необходимо сделать небольшое отступление и рассказать об еще одном способе извлечения текста из OpenOffice.writer, не имеющем отношения ни к работе с библиотеками, ни к технологии отражения [21]. Речь пойдет о ручном разборе контейнера ODT, который, по сути, представляет собой ZIP-архив с файлами данных, описанных в формате XML, а также другими необходимыми файлами описания и объектами документа. Для распаковки архива можно использовать как встроенные средства платформы .Net, так и сторонние библиотеки (например, Ionic.Zip).

Текст, таблицы и ссылки на объекты хранятся внутри расположенного в архиве файла content.xml в некой структуре. Хранение графических объектов осуществляется следующим образом: внутри архива находится папка Pictures, где хранятся сами графические файлы, а ссылка на них, как было сказано ранее, хранится в файле content.xml. Приведем пример ссылки на картинку:

```
<draw:frame draw:style-name="fr2" draw:name="Графический объект1"
text:anchor-type="paragraph"          svg:x="0.037cm"          svg:y="0.201cm"
svg:width="4.217cm" svg:height="3.302cm" draw:z-index="1">

  <draw:image
xlink:href="Pictures/10000000000000CC00000099028356212.jpg"
xlink:type="simple" xlink:show="embed" xlink:actuate="onLoad" />

</draw:frame>
```

Для хранения форматированного текста используется следующая конструкция. В самом начале файла content задаются именованные стили, а затем в тэге, хранящем сам текст, приводится ссылка на этот стиль. В приводимом ниже примере стиль именуется P37.

```
<style:style style:name="P37" style:family="paragraph" style:parent-style-name="Text_20_body">
  <style:text-properties fo:color="#800000" style:font-name="Times New Roman1" fo:font-size="14pt" fo:language="ru" fo:country="RU" fo:font-style="italic" fo:font-weight="bold" /> </style:style>
  ...<text:p text:style-name="P37">В данном задании все выполнено верно (5 баллов)</text:p>
```

Извлечение текста сводится к разбору данного файла, а также при необходимости еще и к разбору файла styles.xml, где описаны стили форматирования данного документа.

Но данный метод не очень удобен в применении, так как нет официальной документации, описывающей структуру архивных файлов.

Мы рассмотрели способы чтения текстовой информации из различных форматов. Теперь рассмотрим реализацию таких дополнительных возможностей, как работа с электронной почтой. Это может быть удобно, если необходимо реализовать автоматическую загрузку анализируемой информации или высылку сформированного отчета.

Работа с электронной почтой

Работа с электронной почтой может потребоваться, например, при решении следующей задачи:

«Разработать систему автоматизации работы юридического консультативного центра. Система должна автоматически регистрировать вопросы клиентов, полученные ею по электронной почте, определять тип вопроса и закреплять его за исполнителем. Должна быть функция автоматического поиска подходящих ответов для каждого вопроса, исходя из схожести полученного вопроса и списка

обработанных, хранящихся в базе данных. Кроме того, должна быть реализована функция отсылки ответа на почту клиента».

Для организации получения программой электронной почты можно воспользоваться возможностями бесплатной библиотеки OpenPop. Это бесплатная .NET библиотека, написанная на С# и полностью реализующая функционал для работы с сервером POP3. Домашняя страница данной библиотеки [22] включает примеры и документацию.

Первое, что нам необходимо сделать – скачать файл OpenPop.dll и подключить его к проекту. Затем прописать пространства имен:

```
using OpenPop.Pop3;  
using System.IO;
```

При этом System.IO нужно для сохранения вложений. Для работы с почтой нам понадобится объект класса Pop3Client:

```
Pop3Client pop3Client = new Pop3Client();
```

Для проверки соединения с сервером можно использовать его свойство Connected, а для отключения от сервера – метод Disconnect(). Следующую конструкцию рекомендуется вставлять в начала кода работы с сервером, чтоб обезопасить себя от устаревших и зависших подключений:

```
if (pop3Client.Connected) pop3Client.Disconnect();
```

Подключение к серверу выполняется методами Connect и Authenticate:

```
pop3Client.Connect(serverName, port, true);  
pop3Client.Authenticate(login, password);
```

В данном случае параметр ServerName – строка с именем сервера, port – число-номер порта подключения, логическое значение Истина говорит об использовании Ssl. Параметры login и password – логин и пароль пользователя.

Метод pop3Client.GetMessageCount() вернет общее количество писем в ящике, а метод pop3Client.GetMessage(<номер сообщения>) –

даст доступ к конкретному письму и вернет объект класса `OpenPop.Mime.Message`. В свойстве `Headers` этого объекта хранится вся нужная информация о письме. Например, пусть текущее сообщение сохранено в переменную `message`, тогда отображаемое имя отправителя можно получить следующей конструкцией:

```
message.Headers.From.DisplayName.ToString()
```

Для получения вложений можно использовать метод объекта-сообщения `FindAllAttachments()`, который вернет список объектов `OpenPop.Mime.MessagePart`. У каждого из его элементов будет важное свойство `FileName` – имя файла-вложения, и метод `Save`, принимающий в качестве параметра файловый поток и сохраняющий в него содержимое вложения. Для удаления сообщения можно воспользоваться конструкцией:

```
pop3Client.DeleteMessage(<номер сообщения>);
```

Рассмотрим пример кода перебирающего все сообщения в ящике, выводящий для каждого информацию о том, от кого это письмо получено, сохраняющий их вложения в указанную папку и удаляющий просмотренное письмо. Считается, что подключение к серверу уже установлено ранее:

```
int count = pop3Client.GetMessageCount();
if (count > 0)
{
    for (int i = count; i >= 1; i -= 1)
    {
        try
        {
            OpenPop.Mime.Message message = pop3Client.GetMessage(i);
            List<OpenPop.Mime.MessagePart> l= message.FindAllAttachments();
            for (int j = 0; j < l.Count; j++)
            {
                MessageBox.Show(l[j].FileName);
                l[j].Save(new
                FileStream(Application.StartupPath+"\\mail\\"+l[j].FileName,
                FileMode.CreateNew));
            }
        }
    }
}
```

```

    }

    MessageBox.Show(message.Headers.From.DisplayName.ToString());
    }

    catch (Exception ex) { MessageBox.Show(ex.Message); }
    pop3Client.DeleteMessage(i);
    }
}

```

Обратите внимание на необходимость конструкции try-catch как при обработке писем, так и при подключении к серверу. Еще следует учесть, что при загрузке больших писем (писем с объемными вложениями) программа может «тормозить».

Для отправки сообщений в сетях TCP/IP предназначен сетевой протокол SMTP (англ. Simple Mail Transfer Protocol — простой протокол передачи почты). Он используется для отправки почты от пользователей к серверам и между серверами для дальнейшей пересылки к получателю.

Данный протокол получил распространение в начале 80-х годов. До этого использовался протокол UUCP, основным недостатком которого была необходимость знания полного маршрута от отправителя до получателя и явного указания этого маршрута в адресе получателя либо наличия прямого коммутируемого или постоянного соединения между компьютерами отправителя и получателя.

Рассмотрим пример реализации отсылки электронного сообщения на языке C#. Во-первых, для доступа к необходимым классам подключим следующие пространства имен:

```
using System.Net.Mail; using System.Net; using System.Net.Mime;
```

Для доступа к серверу создадим объект класса SmtpClient, передав в качестве параметров в конструктор имя сервера и номер порта.

```
SmtpClient Smtp = new SmtpClient("smtp.mail.ru", 25);
```

Теперь укажем в свойстве Credentials логин-пароль для авторизации:

```
Smtp.Credentials = new NetworkCredential("user", "password");
```

Далее перейдем к формированию самого письма, используя объект класса MailMessage.

```
MailMessage Message = new MailMessage();  
Message.From = new MailAddress("userFrom@mail.ru");//от кого  
Message.To.Add(new MailAddress("userTo@mail.ru"));//кому  
Message.Subject = "Проверка связи";//тема  
Message.Body = "Сделано на Си шарп";//текст письма
```

Для прикрепления файла используем объект класса Attachment, добавив его потом в свойство-список объекта MailMessage:

```
string file = Application.StartupPath+"\\reports\\Text.pdf";  
Attachment attach =  
new Attachment(file, MediaTypeNames.Application.Octet);  
Message.Attachments.Add(attach);
```

И собственно отправка письма:

```
Smtplib.Send(Message);
```

Мы рассмотрели работы с электронной почтой. Обратите внимание, предлагаемые методы не являются единственно возможными.

Контрольные вопросы к разделу

1. Перечислите основные объекты MS Word.
2. Перечислите основные объекты OpenOffice.
3. Что такое технология отражения?
4. Перечислите основные объекты, используемые в библиотеке iTextSharp.

Практические задания к разделу

1. Проведите кластерный анализ документа в формате *.doc.
2. Проведите кластерный анализ документа в формате *.odt.
3. Постройте график, визуализирующий результат кластерного анализа и сформируйте отчет о нем в формате pdf.
4. Осуществите загрузку текстов для анализа из электронной почты.

ЗАКЛЮЧЕНИЕ

В данном пособии мы рассмотрели различные алгоритмы анализа слабоструктурированных информационных ресурсов, а также способы их реализации на платформе .NET. В ходе работы над пособием был изучен и переработан большой объем материала, а также изложены авторские разработки последних трех лет. Основной целью работы было освещение базовых и вместе с тем наиболее важных аспектов реализации алгоритмов анализа текстов, которые студентам, возможно, придется использовать в своей будущей работе.

Пособие обобщает наиболее важную информацию о самых часто используемых, а также недавно разработанных алгоритмах анализа слабоструктурированных информационных ресурсов и технологиях реализации этих алгоритмов. Информация представлена в удобной для понимания и использования форме. Все примеры кода являются работающими, поэтому их можно сразу использовать в проектах, но вместе с тем они могут быть оптимизированы и доработаны, что позволит студентам поднять свой профессиональный уровень. Приведенные в библиографическом списке источники литературы ориентируют читателя на более подробное изложение некоторых вопросов.

Приведенные примеры практических задач представляют собой реально существующие проблемы. В ходе работы над ними студенты должны научиться разрабатывать программное обеспечение по анализу текстовой информации. Рассмотренные примеры задач и алгоритмы решения некоторых из них позволяют студентам составить определенное представление о том, что их может ожидать в реальной рабочей практике.

ПРИЛОЖЕНИЕ

Код owl-файла онтологии:

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"

xml:base="http://www.semanticweb.org/vadim/ontologies/2015/7/untitled-
ontology-87"

    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xml="http://www.w3.org/XML/1998/namespace"

ontologyIRI="http://www.semanticweb.org/vadim/ontologies/2015/7/untitl
ed-ontology-87">
    <Prefix name="" IRI="http://www.w3.org/2002/07/owl#" />
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#" />
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-
syntax-ns#" />
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#" />
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-
schema#" />
    <Declaration>
        <Class IRI="#Водоем" />
    </Declaration>
    <Declaration>
        <Class IRI="#Грека" />
    </Declaration>
    <Declaration>
        <Class IRI="#Животное" />
    </Declaration>
    <Declaration>
        <Class IRI="#Пак" />
    </Declaration>
```

```

<Declaration>
    <Class IRI="#Река"/>
</Declaration>
<Declaration>
    <Class IRI="#Человек"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#видит"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#ехалЧерез"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#заРукуЦап"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#находитсяВ"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#сунулРукуВ"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#имеетИмя"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#имеетНазвание"/>
</Declaration>
<Declaration>
    <DataProperty IRI="#имеетНаименование"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#ДревнийГрека"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#ДревнийРак"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Река"/>

```

```

</Declaration>
<SubClassOf>
    <Class IRI="#Грека"/>
    <Class IRI="#Человек"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Рак"/>
    <Class IRI="#Животное"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Река"/>
    <Class IRI="#Водоем"/>
</SubClassOf>
<ClassAssertion>
    <Class IRI="#Грека"/>
    <NamedIndividual IRI="#ДревнийГрека"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Рак"/>
    <NamedIndividual IRI="#ДревнийРак"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Река"/>
    <NamedIndividual IRI="#Река"/>
</ClassAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#видит"/>
    <NamedIndividual IRI="#ДревнийГрека"/>
    <NamedIndividual IRI="#ДревнийРак"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#ехалЧерез"/>
    <NamedIndividual IRI="#ДревнийГрека"/>
    <NamedIndividual IRI="#Река"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#заРукуЦап"/>
    <NamedIndividual IRI="#ДревнийРак"/>

```

```

        <NamedIndividual IRI="#ДревнийГрека"/>
    </ObjectPropertyAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="#находитсяВ"/>
        <NamedIndividual IRI="#ДревнийРак"/>
        <NamedIndividual IRI="#Река"/>
    </ObjectPropertyAssertion>
    <DataPropertyAssertion>
        <DataProperty IRI="#имеетИмя"/>
        <NamedIndividual IRI="#ДревнийГрека"/>
        <Literal datatypeIRI="&xsd:string">Древний
Грека</Literal>
    </DataPropertyAssertion>
    <DataPropertyAssertion>
        <DataProperty IRI="#имеетНаименование"/>
        <NamedIndividual IRI="#ДревнийРак"/>
        <Literal datatypeIRI="&xsd:string">Древний Рак</Literal>
    </DataPropertyAssertion>
    <DataPropertyAssertion>
        <DataProperty IRI="#имеетНазвание"/>
        <NamedIndividual IRI="#Река"/>
        <Literal datatypeIRI="&xsd:string">Река</Literal>
    </DataPropertyAssertion>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#видит"/>
        <Class IRI="#Человек"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#ехалЧерез"/>
        <Class IRI="#Человек"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#заРукуЦап"/>
        <Class IRI="#Животное"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#находитсяВ"/>
        <Class IRI="#Животное"/>
    </ObjectPropertyDomain>

```

```

</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#сунулРукуВ"/>
    <Class IRI="#Человек"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
    <ObjectProperty IRI="#видит"/>
    <Class IRI="#Животное"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#ехалЧерез"/>
    <Class IRI="#Водоем"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#заРукуЦап"/>
    <Class IRI="#Человек"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#находитсяВ"/>
    <Class IRI="#Водоем"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#сунулРукуВ"/>
    <Class IRI="#Водоем"/>
</ObjectPropertyRange>
<DataPropertyDomain>
    <DataProperty IRI="#имеетИмя"/>
    <Class IRI="#Человек"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#имеетНазвание"/>
    <Class IRI="#Водоем"/>
</DataPropertyDomain>
<DataPropertyDomain>
    <DataProperty IRI="#имеетНаименование"/>
    <Class IRI="#Животное"/>
</DataPropertyDomain>
<DataPropertyRange>

```

```

        <DataProperty IRI="#имеетИмя"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#имеетНазвание"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
    <DataPropertyRange>
        <DataProperty IRI="#имеетНаименование"/>
        <Datatype abbreviatedIRI="xsd:string"/>
    </DataPropertyRange>
</Ontology>

<!-- Generated by the OWL API (version 3.3.1957)
http://owlapi.sourceforge.net -->

```

ГЛОССАРИЙ

СИР – слабоструктурированные информационные ресурсы – это первичные ресурсы, не имеющие четко выраженной внутренней организационной структуры. Чаще всего представлены в виде текстовых документов.

САПР – система автоматизированного проектирования – это совокупность аппаратного, программного, организационного обеспечений, предназначенных для максимальной автоматизации процесса проектной деятельности.

Онтология – это формализованное представление предметной области в виде совокупности взаимосвязанных классов, их объектов и аксиом, устанавливающих семантически значимые отношения между её элементами.

Класс (в онтологии) – это группа сущностей предметной области внутри онтологии, обладающих общими свойствами.

Объект (в онтологии) – это элемент класса онтологии, обладающий конкретными значениями свойств.

Морфологический анализ – это определение морфологических характеристик слова/сочетания слов.

Анализ СИР – это извлечение структурированной информации из текста посредством применения соответствующих наборов морфологических, статистических, семантических и др. алгоритмов.

Лемма – это начальная (нормальная) форма слова.

Лемматизация – это приведение слова к начальной форме с учетом его части речи.

Стемминг – это процесс нахождения основы слова.

Термин – это слово или многословие, являющее собой название понятия конкретной предметной области.

Тезаурус – это набор терминов предметной области с описанием семантических отношений между ними. Является упрощенной формой

онтологии в связи с отсутствием аксиом и ограниченным набором типов отношений между терминами.

Стоп-лист – это набор незначимых слов, не участвующих в процессе анализа текста.

Частотный словарь – это набор слов текста со значениями частот их встречаемости в нем.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- CLI-сборка, 125
- ComponentContext, 127
- DataType Properties, 97
- Desktop, 127
- Dictionary, 31
- FCM-кластеризация, 69
- Graphical Device Interface, 49
- Microsoft Word, 120
- Mystem, 57
- Object Properties, 97
- OpenOffice, 125
- OpenPop, 133
- OWL-онтология, 87
- Portable Document Format, 112
- ServiceManager, 126
- SMTP, 135
- System.IO, 53
- XML, 106
- Аксиома, 89
- Делегат, 75
- Диапазон, 98
- Домен, 98
- Класс File, 53
- Класс Graphics, 50
- Класс Math, 18
- Класс StreamReader, 53
- Класс StreamWriter, 53
- Класс String, 22
- Класс StringBuilder, 27
- Класс THING, 95
- Класс Char, 20
- Критерий вложенных связей, 93
- Лемма, 57
- Массивы, 13
- Метод Frequency, 31
- Метод Log-Likelihood, 37
- Метод Mutual Information, 34
- Метод TF*IDF, 38
- Метод T-Score, 37
- Морфологический анализ, 57
- Онтология, 87
- Отражение, 128
- Параметрический класс, 17
- Пространство имен, 8
- Редактор Protégé, 94
- САПР, 5
- Событие, 76
- Стоп-слова, 46
- Тезаурус, 90
- Тезаурусный критерий, 90
- Точка входа, 39

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Maria Teresa Pazienza¹, Marco Pennacchiotti¹, and Fabio Massimo Zanzotto Terminology extraction an analysis of linguistic and statistical approaches // Proceedings of the NEMIS 2004 Final Conference, pp. 255–279.
2. Nenadic G., Ananiadou S., McNaught J. Enhancing Automatic Term Recognition through Variation // Proceedings of 20th Int. Conference on Computational Linguistics COLING'04. 2004. pp. 604–610 [Pecina et.al., 2006; Zhang et.al., 2008].
3. [Электронный ресурс]: Материалы свободной энциклопедии Википедиа. URL: <https://ru.wikipedia.org/wiki/Лямбда-выражения>, дата обращения: 28.08.2015
4. Рыков В. В. Корпус текстов как новый тип словесного единства //Труды Междурнар. семинара, Диалог-2003. М.: Наука. – 2003. – С. 15-23.
5. [Электронный ресурс]: Материалы свободной энциклопедии Википедиа. URL: https://ru.wikipedia.org/wiki/Шумовые_слова, дата обращения: 28.08.2015.
6. [Электронный ресурс]: Документация Mystem. URL: <https://tech.yandex.ru/mystem/doc/index-docpage/>, дата обращения: 28.08.2015.
7. Андреев И.А., Башаев В.А., Клейн В.В. Разработка программного средства для извлечения терминологии из текста на основании морфологических признаков, определяемых программой Mystem // «Интегрированные модели и мягкие вычисления в искусственном интеллекте». – М.: Физматлит. – 2013. с. 1227–1236.
8. Yarushkina N. Soft computing and complex system analysis// International Journal of General Systems. 2001. Vol. 30. № 1. pg. 71-88.
9. Namestnikov A.M., Yarushkina N.G. Efficiency of genetic algorithms for automated design problems// Известия Российской академии наук. Теория и системы управления. 2002. № 2. С. 127-133.

10. Ярушкина Н.Г., Вельмисов А.П., Стецко А.А. Средства data mining для нечетких реляционных серверов данных // Информационные технологии, 2007, № 6, с. 20-29.
11. Митрофанова О.А., Константинова Н.С. Онтологии как системы хранения знаний / Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно - телекоммуникационные системы", 2008. - 54 с.
12. Добров Б.В., Лукашевич Н.В., Сыромятников С.В. Формирование базы терминологических словосочетаний по текстам предметной области // Тр. 5-й Всеросс. научн. конф. «Электронные библиотеки: перспективные методы и технологии, электронные коллекции» (RCDL-2003). – СПб. 2003. с. 201–210.
13. Афанасьева Т.В., Ярушкина Н.Г. Нечеткий динамический процесс с нечеткими тенденциями в анализе временных рядов // Вестник Ростовского государственного университета путей сообщения, 2011, № 3, с 7-16.
14. Кураленок И.Е., Некрестьянов И.С. Оценка систем текстового поиска. Программирование. - 28(4). – 2002. - С.226-242.
15. [Электронный ресурс]: Материалы свободной энциклопедии Википедиа. URL: <https://ru.wikipedia.org/wiki/XML>, дата обращения: 28.08.2015.
16. [Электронный ресурс]: Материалы сайта «ITextSharp» . URL: <http://itextpdf.com> , дата обращения: 28.08.2015
17. [Электронный ресурс]: Материалы свободной энциклопедии Википедиа. URL: https://ru.wikipedia.org/wiki/Microsoft_Word, дата обращения: 28.08.2015.
18. [Электронный ресурс]: Молчанов, В. Работа с серверами автоматизации Word и Excel в Visual Studio .Net. / В. Молчанов. URL: http://wladm.narod.ru/C_Sharp/componentbegin.html, (дата обращения: 28.08.2015

19. [Электронный ресурс]: Чернов, Д. Работа с OpenOffice на C#. / Д. Чернов. URL: <http://life-sat.blogspot.ru/2007/06/openoffice-c-net.html>, дата обращения: 28.08.2015.

20. [Электронный ресурс]: Материалы SQL-форума. URL: <http://www.sql.ru/forum/415013/c-openoffice>, дата обращения: 28.08.2015.

21. Воронина, В. В. Разработка pdf-конвертера документов OpenOffice.writer / В. В. Воронина, Е. В. Дементьев // Вузовская наука в современных условиях : сборник материалов 47-й научно-технической конференции. Ч 2. – Ульяновск : УлГТУ, 2013. – с. 228–230

22. [Электронный ресурс]: Документация библиотеки OpenPop. URL: <http://hpop.sourceforge.net>, дата обращения: 28.08.2015.

Учебное издание

ВОРОНИНА Валерия Вадимовна
МОШКИН Вадим Сергеевич

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ АНАЛИЗА СЛАБОСТРУКТУРИРОВАННЫХ ИНФОРМАЦИОННЫХ РЕСУРСОВ

Учебное пособие

Редактор А. В. Ганина

Подписано в печать __.__.2015.

Бумага писчая. Усл.печ.л.

Тираж 100 экз. Заказ

Ульяновский государственный технический университет,
432027, г. Ульяновск, Сев. Венец, д.32.

Типография УлГТУ, 432027, г. Ульяновск, Сев. Венец, д.32.