Софийски университет
"Св. Климент Охридски"
Факултет по математика и информатика

Дипломна работа

Синтактичен и семантичен анализатор на свободно изписани пощенски адреси

Даниела Василева Славкова
Магистърска програма: "Разпределени системи и мобилни технологии"
Катедра: "Информационни технологии"
Фак.№: M21301
Научен ръководител: доц. д-р. Тинко Тинчев

1 Съдържание

2 Увод

Съвременният компютърен свят е почти неограничен. Бизнесът днес се крепи на различни платформи и работи с разнообразни технологии. Все повече се увеличава и зависимостта на обществото от компютърната техника и високите технологии, защото те намират значително място в нормалния живот и нужди на хората. Наистина изключителна е ролята, която компютрите и софтуерът заемат в съвременното общество. В ерата на дигиталните комуникации и развитите технологии, използването на компютърни програми е на практика неограничено.

Развитието на съвременните технологии подтиква разработката на всякакъв вид приложения, които да улесняват ежедневния живот на хората, просто като ги превърнат в потребители на тези приложения.

2.1 Цели и обхват на разработката

Българските карти са уникална интернет услуга. Те предлагат географски, административни и транспортни карти на България и големите градове - всяко населено място с точното му местоположение. Търсене по име или пощенски код е една ценна възможност, която спестява много време и усилия. Намирането на всяка една улица или квартал, или по-общо, на всеки един адрес в София е голямо удобство.

В практиката се наблюдава, че при въвеждането в компютър на пощенски адрес не се спазва реда на попълване и прецизно изписване на имената на селища, квартали, улици и другите съставни части. Това пречи на по-нататъшното им анализиране и разпознаване. Например, в адреса "гр. София, бул. Ген. Михаил Скобелев 67" са възможни редица съкращения и/или (правописни) грешки при изписването. Съвсем реален е вариантът "гр. София, ул. Скобилев 67". От друга страна, анализът на адреса е препятстван от размера на базата от данни за претърсване с номенклатурите на действителните адреси. Възможността за неволна/умишлена смяна на входния език също не е изключена и довежда до усложнения ('с' на латиница и 'с' на кирилица например имат различни ASCII кодове, но еднакво графично представяне). Следователно система, която от една страна потвърждава дали даден адрес съществува, а от друга, ако не съществува, какви са най-добрите приближения, има практически смисъл.

Цел на дипломната работа е да се разработят алгоритъм и реализираща го програмна система, такива че при направен вече анализ на съставните части на изписания адрес, да се нормализира адресът и да се предложи правилно изписване на същия. Това означава, че при направена заявка за улица или квартал, например, системата трябва да върне най-подходящия кандидат за тази улица или за този квартал. Реализацията има

за цел да нормализира отделните части на адреса сами за себе си, като самостоятелни единици.

Инициативата за решаването на задачата, която е основа за дипломната работа, е на фирма DataMap, която професионално се занимава с въпроси, свързани с определяне на географското местоположение на обект. Тя е реализирала десетки Географски Информационни Системи (ГИС проекти) в България. Един от основните й продукти е MapInfo Professional, който съчетава основните възможности на CAD, СУБД и електронните таблици, взети заедно. Истинската сила на продукта е в едновременното поддържане и анализ на географска и таблична информация. MapInfo е най-масовата ГИС заради лекотата, с която се усвоява от потребителите. Тя предоставя силата на географските анализи на широк кръг потребители.

Един специален случай на географски анализ е намирането на информация за заявен от потребител адрес. Именно в този случай се появява нуждата от определянето на това дали адресът е действителен или не (дали съставните му части са действителни географски обекти или не), като във втория случай възниква по същество проблемът: да се намерят най-подходящите, съществуващи в налична база от данни адреси, един от които вероятно е имал предвид потребителят.

Обхватът на решението е ориентиран да отговаря на практическите нужди на фирмата възложител. Първоначалната задача е да се предостави на фирмата възложител решение, с което тя да се запознае. Чрез представяне на стабилна функционалност за работа и предоставянето и на фирмата, най-бързо може да се провери коректността на решението и да се получат препоръките и корекциите в най-ранен стадии. Идеята е да се предостави решение, което да бъде оптимално при използването му.

2.2 Структура

Разработката представлява сървър приложение, което осъществява достъпа до базата от данни с действителните адреси. То изпълнява задачата за бързо обхождане и претърсване на данните.

Решението логически е разделено на модули, които си взаимодействат. Разделението е на базата на отделните функционалности, които трябва да обхване цялостната реализация. Модулите, които сървърът обединява са следните:

- Модул FdaObj – декларация и описание на интерфейса на сървър приложението;
- Модул Alphabet – декларация на входната азбука, от която се състоят всички думи в базата и структури за бързия достъп до буквите й;

- Модул Algorithm - директно построяване на минимален краен детерминиран автомат от списък с думите във входната база данни;
- Модул DataBase – достъп до входната базата от данни;
- Модул RequestDictionary – осъществява претърсване на речника и намира най-подходящите кандидати;
- Модул Levenshtein – алгоритъм за намиране на най-близките кандидати в смисъл на Левенщайн;
- Модул ProcessingErrors – обработва грешките като съобщава за наличието на определен тип грешка в изпълнението на сървъра.

3 Използвани модели, спецификации и технологии

Изискванията на фирмата възложител на задачата, която е основата за дипломната работа, относно технологиите, които трябва да бъдат използвани при реализацията, са разглеждани като ограничаващи/облекчаващи условия. Тези условия са следните:

- езика за програмиране, на който трябва да бъде реализирана разработката, е Visual C++ или Visual Basic;
- номенклатурите се поддържат в Access база данни – MDB.

Тъй като системата, естествено, трябва да е съгласувана с наличната технология във фирмата възложител на задачата, по-горните условия са спазени.

3.1 Избор на програмен език и среда на разработка

Факторите които повлияха при избора на Microsoft Visual C++ за среда на разработка на приложението са няколко.

Един от основните, но не на последно място, критерий е простотата в характера на C++. При създаването на C++ е важало следното правило: когато може да се избере по-просто дефиниране или по-прост компилатор, се избира първото. В C++ са избягвани средства, които могат да предизвикат проблеми при изпълнение или проблеми с паметта, дори и без да бъдат използвани. Примерно в C++ са били отхвърлени конструкции, които изискват във всеки обект да се записва информация за вътрешното разполагане на данните. Затова ако програмистът декларира структура, състояща се от две 16-битови величини, тази структура може да се разположи в един 32-битов регистър.[2]

C++ е проектиран за използване в традиционна среда за компилиране и изпълнение – в C програмна среда върху UNIX система. Но той не е ограничен само до UNIX, а просто използва UNIX и C като модел за взаимовръзка между език, библиотеки, компилатори, свързващи програми, среда за изпълнение и т.н. Този минимален модел позволява на C++ да работи върху почти всяка платформа.[2]

Друг основен момент при избора на език за програмиране е структурата на програмите му. Ударението върху структурата в C++ отразява нарастването на програмите. Малка програма (примерно до 1 000 реда) може да бъде подкарана с груба сила, дори да бъдат нарушени всички правила на добрия стил на програмиране. При големите програми това просто е невъзможно. Ако структурата на програма от 100 000 реда е лоша, новите грешки се появяват със скоростта на отстраняване на старите. C++ е проектиран така, че големите програми да могат да се

структурират по рационален начин, за да може сам човек да се справи с огромното количество код.

Един от най-съществените критерии, които предшестваха избора на C++, е и неговата стандартна библиотека. Мечтата на програмиста е да намери в библиотека всеки интересен, значим и сравнително общ клас, функция, шаблон и т.н. Стандартната библиотека е нещо, което всяка реализация трябва да съдържа и всеки програмист да може да използва. Стандартната библиотека на C++:

- Поддържа средствата на езика, например управление на паметта и идентификация на типа по време на изпълнение;
- Предоставя функции, които не могат да бъдат реализирани оптимално чрез самия език за всяка система, например sqrt(), memmove();
- Предоставя средства, за които програмистът може да разчита, че са преносими, например списъци, карти(map), сортиращи функции и входно/изходни потоци;
- Дава рамка за разширяване на средствата, които предоставя, например конвенции и поддържащи средства, благодарение на които потребителят може да предостави вход/изход на потребителските типове, аналогичен на входа/изхода на вградените типове;
- Дава общата база за всички други библиотеки.

Дизайнът на библиотеката е определен главно от последните и три роли. Тези роли са тясно свързани. Например, преносимостта е важен критерий при проектирането на всяка специализирана библиотека, а общите контейнерни типове, като списъци и карти, са необходими за удобната комуникация между отделно създаваните библиотеки.

Последната роля е особено важна от гледна точка на проектирането. Например, средствата низ и списък се предоставят от стандартната библиотека. Ако тя не ги предоставяше, създадените независимо една от друга библиотеки щяха да комуникират единствено чрез вградени типове. Обаче, графични средства или търсене на регулярен израз не се предоставят. Тези средства се използват много широко, но те рядко участват директно в комуникацията между независимо създадени библиотеки.

Средствата, предлагани от стандартната библиотека на C++ са:

- Ценни за и използваеми от практически всеки студент и професионалист;
- Използвани директно или индиректно от всеки програмист за всяко нещо, което е в обхвата на библиотеката;
- Достатъчно ефективни, за да бъдат наистина алтернатива на ръчно написаните функции, класове и шаблони;
- Примитивни (в математическия смисъл) – индивидуални компоненти, предназначени за една единствена роля. Всеки

компонент, който се явява в две слабо свързани роли почти със сигурност ще бъде по-неефективен;
- Удобни, ефикасни и сравнително сигурни за обща употреба;
- Завършени във функцията си;
- Съчетават и разширяват вградените типове и операции;
- По подразбиране контролират и гарантират типовете;
- Поддържат разпространените стилове на програмиране;
- Могат да бъдат разширяеми така, че да работят по същия начин с потребителските типове, както работят с вградените типове на стандартната библиотека.

C++ се използва от 1983г. От тогава насам са дефинирани няколко версии и са се появили много независимо създадени реализации. Главната цел на стандартизирането е била да помогне на реализаторите и потребителите да разполагат с единна дефиниция на C++.

Следващ критерий е оптималност и надеждност на генерираният изпълним код на приложението. Със своя добре реализиран компилатор от 1991г., който е претърпял, от създаването си до сега, само някои минимални промени, Microsoft Visual C++ е лидер по отношение на скорост на изпълнение на създаваните изпълними приложения и безпроблемно използване върху Windows платформи.

3.2 Избор на технология

За всеки компонент, който се програмира, трябва да се проектира програмен модел. Това означава, че трябва да се вземе решение как компонента ще бъде програмиран или, с други думи, как трябва да се напише кода за манипулиране на този компонент.

Терминът програмен модел е често наричан още обектен модел. Под програмен модел се разбира множеството от интерфейси, свойства, методи и събития, предоставени от един компонент, което позволява да бъдат написани програми, които да го манипулират.

Проектирането на добър програмен модел е точно толкова важно, колкото и проектирането на добър потребителски интерфейс. Много от принципите, използвани в проектирането на потребителския интерфейс, могат да бъдат приложени директно върху проектирането на програмния модел. Добрият потребителски интерфейс позволява на потребителите да работят с най-високото ниво на абстракция без да е необходимо да се занимават с вътрешната имплементация. Потребителският интерфейс представя логическия изглед на функционалността, което е обратното на физическата същност на тази функционалност. Той изразява нещата по начин, който съвпада с това как мисли потребителят, не непременно как работи системата. Потребителският интерфейс,

в комбинация с това как той реагира на потребителското взаимодействие, често е наричан “потребителски модел”.

По същия начин един добър програмен модел не просто трябва да изложи вътрешната си структура – той трябва да изложи функционалността си на по-високо ниво на абстракция така че потребителят (програмистът) да може да се концентрира върху това какво иска да направи, а не върху това как да осъществи една проста задача. Основният момент тук е как другите програмисти ще конструират цялостни програми, използвайки този компонент.

Едно от желанията на фирмата възложител на задачата беше системата да бъде реализирана по един от следните два начина:

- Конзолно приложение, което приема от командния ред аргументите: .mdb файл, таблица и колона от таблица, която съдържа списъка от съответните географски обекти – улици/квартали/..., сред които ще се търси заявената улица/квартал и нейните приближения, и файл, който ще съдържа изходните резултати;
- COM компонента, която има същите входни параметри: .mdb база, таблица от тази бази, колона от таблицата и файл за изходните резултати.

Изборът, който е направен за реализацията на поставената задача, е програмирането на COM обект. Този избор е основан на характеристиките на COM.

Component Object Model (COM) е софтуерна архитектура, която позволява компоненти, направени от различни софтуерни производители, да бъдат комбинирани в разнообразие от приложения. COM дефинира стандарт за компонентна interoperability – възможност една система или продукт да работят с други системи без да са необходими специални средства дори и да са направени от различни производители. COM компонентът не зависи от езика за програмиране. Той може да бъде използван на много платформи и е разширяем.

COM е компонентна софтуерна архитектура, която:

- Дефинира бинарен стандарт за компонентна interoperability;
- Независима е от езика за програмиране;
- Приложима е за много платформи (Microsoft Windows, Microsoft Windows NT, Apple Macintosh, UNIX);
- Осигурява категорична еволюция на компонентно-базираните приложения и системи;
- Разширяема е.

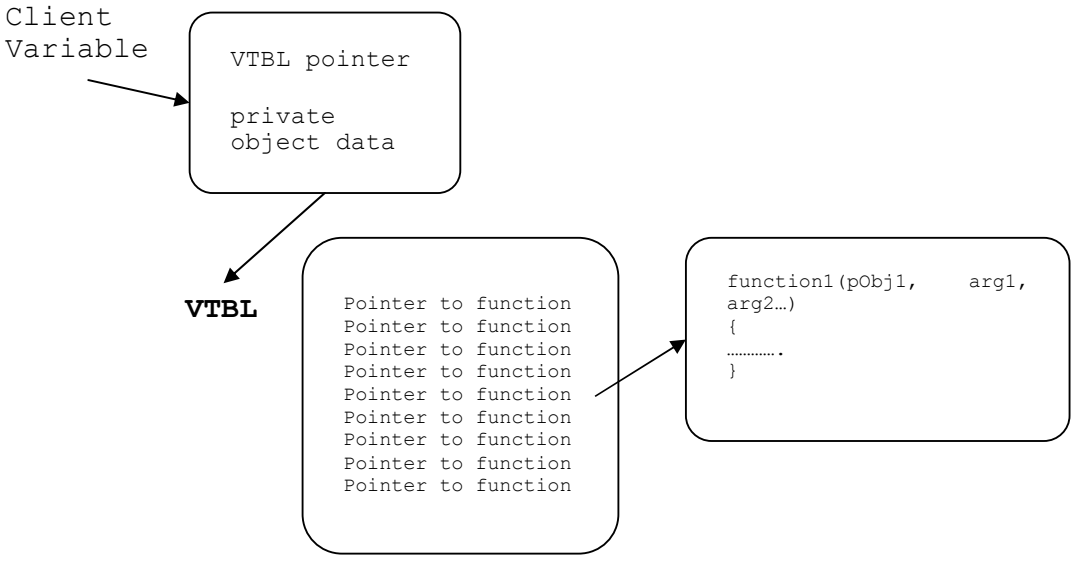
В добавка COM осигурява механизми за следните неща:

- Комуникация между компонентите, дори извън границите на процеса и мрежата;
- Управление на поделената памет между компонентите;

- Съобщения за статуса и грешките;
- Динамично зареждане на компонентите.

COM е основна архитектура за компонентен софтуер. Това, което го прави толкова използван и унифициран модел е, че COM дефинира няколко основни принципа на проектиране и архитектурни концепции, които осигуряват структурните основи на модела. Те включват:

- Бинарен(двоичен) стандарт за извикването на функциите между компонентите;
- Предоставяне на строго-типизирана група от функции в интерфейсите;
- Базов интерфейс, осигуряващ:
  а) Начин компонентите динамично да откриват интерфейсите, имплементирани от други компоненти;
  в) Средство, което позволява компонентите да оставят следа за тяхното време на живот и да се изтриват когато трябва;
- Механизъм, който уникално идентифицира компонентите и техните интерфейси.



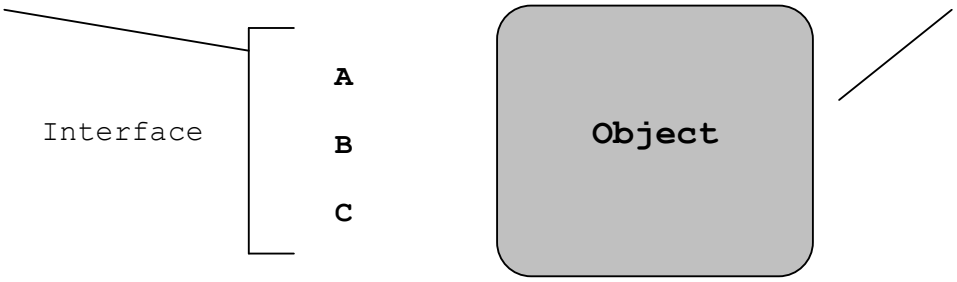
Фигура 1. Диаграма на vtable

За дадена платформа (хардуер или операционна система), COM дефинира стандартен начин да постави таблицата на виртуалните функции (vtables) в паметта, и стандартен начин да извика функциите чрез vtables. Така всеки език, който може да вика функции чрез указатели (C, C++, Small Talk, Ada и дори Basic), може да бъде използван да се напише компонент, който може да взаимодейства с други компоненти, написани на същият бинарен стандарт. Двойната насоченост (клиентът държи указател към

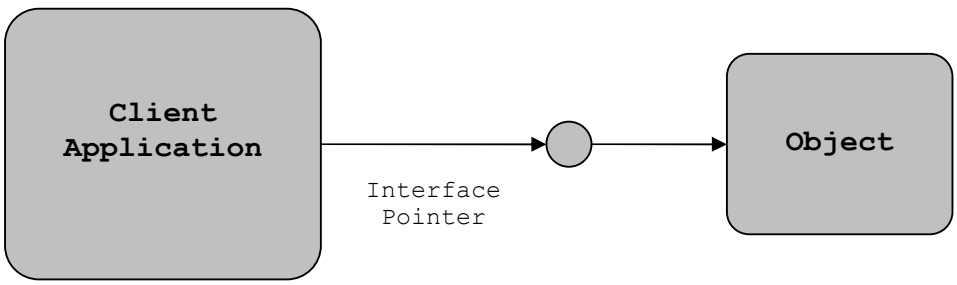
указател към vtable) позволява поделянето на vtable между много инстанции на един и същи обектен клас. В системи със стотици обектни инстанции, vtable поделянето може да спести значително използването на паметта.

Думата обект означава различни неща в различните случаи. В COM обектът е част от компилиран код, който осигурява някаква услуга за останалата част от системата. За да бъде избегнато объркването най-добре е един COM обект да се свързва с "компонентен обект" или просто с "компонент". Компонентните обекти поддържат базов интерфейс наречен IUnknown заедно с комбинация от други интерфейси, зависещи от това каква функционалност избира компонента да изложи.

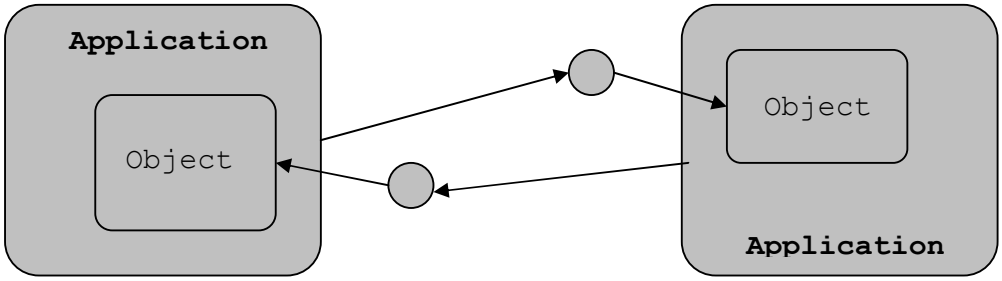
Компонентните обекти обикновено са асоциирани с някакви данни, но не както C++ обектите. Даденият компонентен обект никога няма директен достъп до друг компонентен обект в неговата цялост. Вместо това компонентните обекти винаги достъпват други компонентни обекти чрез интерфейсни указатели. Това е първичната архитектурна характеристика на Component Object Model, защото тя позволява COM напълно да запази капсулацията на данните и процеса - едно от основните изисквания на истинския компонентен софтуерен стандарт.



Фигура 2. Типична картина на компонентен обект който поддържа три интерфейса A, B и C



Фигура 3. Интерфейсите се разширяват по посока на клиента свързан към тях



Фигура 4. Две приложения могат да бъдат свързани едно към друго чрез обекти, в който случай те простират техните интерфейси към обекта на другото приложение

С release-а на Visual C++ версия 2.1, създаването на COM обект, базиран на MFC класове става много лесно. COM е вид "обектно-ориентирано приближение" на DLL-ите. COM обектите са по гъвкави от Win32 DLL-ите, защото те са изцяло езиково независими и лесно се напасват в дизайна на обектно-ориентирани програми.

Традиционните операционни системи тълкуват единствено приложенията като бинарни, но не и компонентите. Дълго време операционните системи са били свързвани само с бинарни приложения като EXE-тата. Обектите в един процес не могат да комуникират с обектите в друг процес използвайки техните собствени дефинирани методи. Операционната система дефинира сигурен механизъм на междупроцесна комуникация като DDE, TCP/IP, Sockets, memory-mapped I/O или именовани канали. Тези обекти е необходимо да използват дефинираните от операционната система механизми, за да комуникират един с друг.

Компонентите, програмирани използвайки COM на Microsoft, осигуряват начин чрез който два обекта в различни обектни пространства или мрежи, могат да говорят помежду си чрез извикването на методите на другия. Тази технология заставя операционната система да види приложенията като обекти.

COM заставя операционната система да действа като централен регистър за обектите. Операционната система поема отговорността за създадените обекти когато те го изискват, изтрива ги когато те не са били изтрити и поддържа комуникацията между тях, независимо дали са в същия процес или на същата машина. Едно основно предимство на този механизъм е, че ако COM обектът някога е бил обновен до нова версия, приложението което използва този обект не е необходимо да бъде прекомпилирано.

Едно от хубавите неща на COM обектите е, че те никога не са свързани специално към някое приложение. Единственото нещо което приложението може да знае за COM обекта е какви функции поддържа или не поддържа. Обектния модел е доста гъвкав, защото приложенията могат по време на изпълнение, да питат COM обекта каква функционалност осигурява. Друго важно предимство на COM технологията е, че когато няма останали указатели към обект, той се самоунищожава.

COM поддържа „Marshalling". „Marshalling" е процес на пакетиране и предаване на данни между различни адресни пространства, който автоматично решава проблемите с указателите, запазвайки целостта и вида на данните. Дори когато COM обектите изпълняват различни процеси или дори се изпълняват на различни машини, операционната система се грижи за „Marshalling".

COM е пуснат през 1993 с OLE2 предимно, за да замени механизма за комуникации между процесите DDE, използван от началните версии на OLE. ActiveX също е базирана на COM. Целия скелет на Windows NT е писан с използването на COM. Той е основата на редица услуги в Windows като COM+, Distributed COM (DCOM), MSMQ, and ActiveX контролите.

3.3 Избор на достъп до базата от данни

Едно от изискванията (условията) на фирмата възложител на задачата DataMap е, че номенклатурите с имената на улиците, кварталите и жилищните комплекси се съхраняват в MDB база от данни.

MFC поддържа два различни вида достъп до база от данни:

- чрез Data Access Object (DAO) и Microsoft Jet database engine;
- чрез Open Database Connectivity (ODBC) и ODBC драйвер.

И двата метода притежават абстракции, които опростяват работата с бази от данни и се изпълняват бързо, мощно и гъвкаво за C++. И двата метода интегрират работата с базата от данни с MFC библиотеките. Двете множества от MFC класове позволяват да достъпваме широко разнообразие от сорсове от данни и правят възможно написването на приложения, които са независими от сорса на данните.

Базите от данни, които могат да се достъпват, използвайки DAO и MFC DAO класовете са:

- Бази от данни, използващи Microsoft Jet database engine, създаден с Microsoft Access или Microsoft Visual Basic, версии 1.x, 2.x, и 3.0 на database engine;
- Инсталируеми ISAM бази от данни, включващи: Microsoft FoxPro, версии 2.0, 2.5, и 2.6.(могат да се експортват и

импортват данни от и в версии 3.0, но не могат да се създават обекти); dBASE III, dBASE IV, и dBASE 5.0; Paradox, версии 3.x, 4.x, и 5.x;
- Open Database Connectivity (ODBC) бази от данни, включващи, но не ограничени от, Microsoft SQL Server, SYBASE SQL Server и ORACLE Server. За да бъде достъпена една ODBC база от данни трябва да е наличен подходящ ODBC драйвер за базата от данни, която се достъпва;
- Microsoft Excel, версии 3.0, 4.0, 5.0 и 7.0 worksheet-и;
- Lotus WKS, WK1, WK3 и WK4 spreadsheet-и;
- Текстови файлове.

DAO е най-добре да се използва с бази от данни, които Microsoft Jet database engine може да чете. Това включва всички изброени по-горе с изключение на ODBC сорсовете от данни. Изпълнението е най-добро с Microsoft Jet (.MDB) бази от данни.

Използвайки ODBC и MFC ODBC класовете може да се достъпват сорсове от данни, локални или отдалечени, за които потребителя на приложението има ODBC драйвер. За широк кръг от сорсове от данни са налични 16-битови и 32-битови ODBC драйвери. Ако се работи с Microsoft Jet (.MDB) база от данни, по-ефикасно е да се използват DAO класовете отколкото Microsoft Access ODBC драйверите.

Обобщено, при избора на ODBC или DAO трябва да се обърне внимание на следните неща:
- Трябва да се използват ODBC класове ако се работи точно с ODBC сорсове от данни.
- Трябва да се използват DAO класовете ако се работи точно с Microsoft Jet (.MDB) бази от данни или с други формати бази от данни, които Microsoft Jet database engine може да чете директно.
- ODBC сорсове от данни могат да се достъпват чрез DAO класове, когато е желателна скоростта на Microsoft Jet database engine и екстра функционалността на DAO класовете.
- DAO изисква допълнително дисково пространство.
- Възможностите на ODBC драйверите варират.

DAO класовете имат следните предимства:
- По-добро изпълнение в някои случаи, особено когато се използва Microsoft Jet (.MDB) база от данни;
- Съвместимост с ODBC класовете и с Microsoft Access Basic и Microsoft Visual Basic;
- Възможност да се специфицират взаимоотношения между таблиците;
- По-богат модел за достъп до бази от данни благодарение на поддръжката на Data Definition Language (DDL) така добре както и Data Manipulation Language (DML).

Всички тези факти предпоставят избора на MFC DAO класовете като начин за достъпване до базата, от която желаната програмна система трябва да чете своите данни.

В следващата таблица е предложено обобщение на ключовите разлики между DAO и ODBC класовете:

Мога ли да ...	**с DAO класове?**	**с ODBC класове?**
Достъп до .MDB файлове	Да	Да
Достъп до ODBC сорсове от данни	Да	Да
Наличност за 16 Bit	No	Да
Наличност за 32 Bit	Да	Да
Database engine поддръжка	Microsoft Jet database engine	Target DBMS
DDL поддръжка	Да	Само чрез директни ODBC извиквания
DML поддръжка	Да	Да
Природа на MFC имплементацията	“Обвивка” на основните функции на DAO	По-скоро опростява абстракцията отколкото да е “обвивка” на ODBC API
Оптимален за	.MDB файлове (Microsoft Access)	Всяка DBMS, за която има наличност на драйвери

Таблица 1. Обобщение на ключовите разлики между DAO и ODBC класовете

4 Анализ и проект на разработката

4.1 Описание на желаната реализация

Както беше уточнено по-горе инициативата за решаването на задачата, която е основата за дипломната работа, е на фирмата DataMap, която професионално решава въпроси, свързани с Географските Информационни Системи. По същество проблемът на задачата е свързан с намирането на информация за заявен от потребителя адрес, или казано с други думи определянето на това дали адресът е действителен или не. Ако адресът (улица, квартал или т.н.) е недействителен, трябва да се намерят най-подходящите кандидати за този адрес, съществуващи в налична база от данни, някой от които вероятно е имал предвид потребителят. Например в адреса "гр. София, бул. Ген. Михаил Скобелев 67" са възможни редица съкращения и/или (правописни) грешки при изписването. Съвсем реален е вариантът "гр. софия, ул. Скобилев 67".

Основните въпроси при разрешаването на поставената задача са няколко. На първо място по ред на изпълнение стои проблема за намирането на алгоритъм за бързо обхождане и претърсване на базата данни с действителните адреси и едновременно с това колекциониране на подходящи кандидати, в случай на недействителен адрес. Тъй като размерът на базата от данни в общия случай е голям, последователното преминаване по записите и евентуалният анализ на приближенията ще забавя много изпълнението на програмата, а и ще трябва да се прави всеки път при всяка заявка. Този подход във всеки случай прави приложението тромаво и практически неприложимо. Затова изборът, на който се уповава решението на тази задача, е чрез еднократно построяване на речник от всички пощенски адреси в базата от данни (по-точно няколко речника за всяка от частите на пощенския адрес като улица, квартал и други) във вид на минимален детерминиран автомат и зареждането му в паметта на компютъра, което позволява разумно бързо обхождане на базата от данни.

Едновременно с това на преден план стои въпроса при обхождането да се колекционират близките до заявката записи от базата. При изписване на заявката са вероятни множество правописни грешки. Възможността за неволна/умишлена смяна на входния език също е реална и поставя нови специфични проблеми. Изобщо, изгледите за грешки са с твърде разнообразна природа. Намирането на подходящи кандидати е един от водещите проблеми при решаването на задачата.

Под близост на два пощенски адреса (две улици или два квартала), което е частен случай на близост на две думи, се разбира до колко едната дума прилича на другата дума (до колко едната дума съвпада с другата дума по начин на изписване). Този проблем в практиката е познат като "approximate pattern

matching”. Въпросът с избора на мярката за близост между думи е съществен, тъй като близките по отношение на нея адреси трябва да са реални кандидати. Разбира се, изчислителната сложност на алгоритъма трябва да е напълно приемлива за практическо използване. При тези забележки близост в смисъл на Левенщайн има действително приложение и изглежда достатъчно подходящ.

След този обзорен преглед на функционалността на желаното решението на преден план излиза въпросът с избора на подходящи структури от данни за реализация на разработените или избраните измежду наличните алгоритми и написването на програмна система, отговаряща на заявката на фирмата и наложилите се в практиката стандарти.

C++ е програмен език с общо предназначение, подходящ за системно програмиране, който

- поддържа абстракция на данните,
- поддържа обектно-ориентирано програмиране,
- поддържа родово програмиране.

За език казваме, че поддържа определен стил на програмиране, ако предоставя средства за удобно (сравнително лесно, сигурно и ефективно) използване на този стил. Един език не поддържа определен подход, ако изисква извънредни усилия или умения, за да бъде написана такава програма; той просто позволява използването на този подход. Например, могат да се напишат структурирани програми на Fortran77 и обектно-ориентирани програми на C, но това е изключително трудно, понеже тези езици не поддържат директно тези подходи.

Сега ще засегнем някои стилове на програмираме, за да изясним употребата им в предложеното решение на задачата. Представянето преминава през последователност от методи, започвайки от процедурното програмиране и води към използването на йерархии от класове в обектно-ориентираното и родово програмиране чрез шаблони. Всяка парадигма надгражда предшествениците си, всяка добавя нещо ново към инструментариума на програмиста и всяка описва един утвърден подход на програмиране. [2]

Оригиналната парадигма на процедурното програмиране е: *Реши от какви процедури се нуждаеш; използвай най-добрите алгоритми, които можеш да намериш*. Ударението е върху обработването – алгоритъмът на изчислението.

Последователно ударението се премества от структурирането на процедурите към организацията на данните. Съвкупността от свързаните процедури и данни често се нарича модул. Парадигмата на модулното програмиране е: *Реши от какви модули се нуждаеш; раздели програмата на части така че данните да бъдат скрити в модули*. Тази парадигма е известна и като

принцип на скритите данни. Там където процедурите и свързаните с тях данни не образуват групи, е приложим и стилът процедурно програмиране. Похватът "добри процедури" вече се прилага към всяка процедура от модула. Принципът на скриване на данните може да се разшири до "скриване на информацията", т.е. имената на функциите, типовете и т.н. също могат да бъдат направени локални за модула.

Модулността е фундаментален аспект на всички добри големи програми. Обаче модулите, във вида описан дотук, не са подходящи за отразяване на сложни системи. Програмирането чрез модули води до централизирането на всички данни за един тип под контрола на модула. Потребителските типове, реализирани чрез модул, предоставящ достъп до типа на реализацията, не се държат като вградени типове и имат по-слаба поддръжка от страна на езика. C++ решава този проблем като позволява на потребителя да дефинира типове, чието поведение е почти същото като на вградените типове. Такъв тип често бива наричан тип абстрактни данни(тип, дефиниран от потребителя). Парадигмата на програмирането с потребителски дефинирани типове е: *Реши от какви типове се нуждаеш; създай пълен набор от операции за всеки тип*. Когато няма нужда от повече от един обект от определен тип, програмният стил на скриване на данните чрез модули върши работа.

Потребителят може да дефинира типове според нуждите си. Тези типове се наричат конкретни типове. При тях представянето на данните му не е отделено от интерфейса, напротив – то е част от кода, който ще се включи към програмния фрагмент. Може структурата да е private (и следователно е достъпна само чрез функциите-членове), но тя е там. Конкретните типове се различават от абстрактните типове, в които интерфейса по-силно изолира потребителя от детайлите на реализирането им. Те изолират потребителя напълно от промените в реализацията му. При абстрактните типове интерфейса е отделен от структурата и няма истински локални променливи. Това става благодарение на виртуалните функции.

Абстракцията на данните е фундаментална за обектно-ориентираното програмиране. Неговата парадигма е следната: *Реши от какви класове се нуждаеш, задай пълен набор от операции за всеки клас, изрази родството им чрез наследяване*. Където не съществува такова родство, достатъчна е само абстракцията на данните. Степента на родство на класовете, което може да се изрази чрез наследяване и виртуални функции, е критерий за приложимостта на обектно-ориентираното програмиране към конкретния проблем. В някои области, като интерактивната графика, има обширно пространство за обектно-ориентирано програмиране. В други области, като класическите аритметични типове и изчисления, базирани на тях, изглежда излишно прилагането на нещо повече от абстракция на данните и не е необходима поддръжка на обектно-ориентирано програмиране.

По принцип концепцията за абстрактен тип данни е обща и е независима от идеята за символите. Следователно двете концепции трябва да бъдат представени независимо една от друга. По-общо казано, ако един алгоритъм може да бъде изразен независимо от детайлите по реализирането му, това е за предпочитане (родово програмиране). Парадигмата на родовото програмиране е: *Реши какви алгоритми искаш, параметризирай ги така че да работят с разнообразие от подходящи типове и структури данни*. Контейнерите са типичния пример за родово програмиране. Стандартната библиотека на C++ предоставя разнообразни контейнери. Следователно може да се приложи парадигмата на родовото програмиране, за да се параметризират алгоритмите чрез контейнери. Примерно ако е необходимо да бъде направено сортиране, копиране и претърсване на вектори, списъци и масиви, без да се налага да се пишат функции sort(), copy() и search() за всеки контейнер.

След преглед на стиловете за програмиране и потребителските изисквания става ясно, че най-подходящата организация на функционалността е модулната, надградена с потребителски дефинирани типове. Това е така, защото от една страна съществува изискване разработката на приложението да е максимално улеснена и време спестяваща, а от друга страна съществува необходимостта от по-добра и по-лесна ориентация в системата, което в по-голяма степен да гарантира коректността на алгоритмичните реализации.

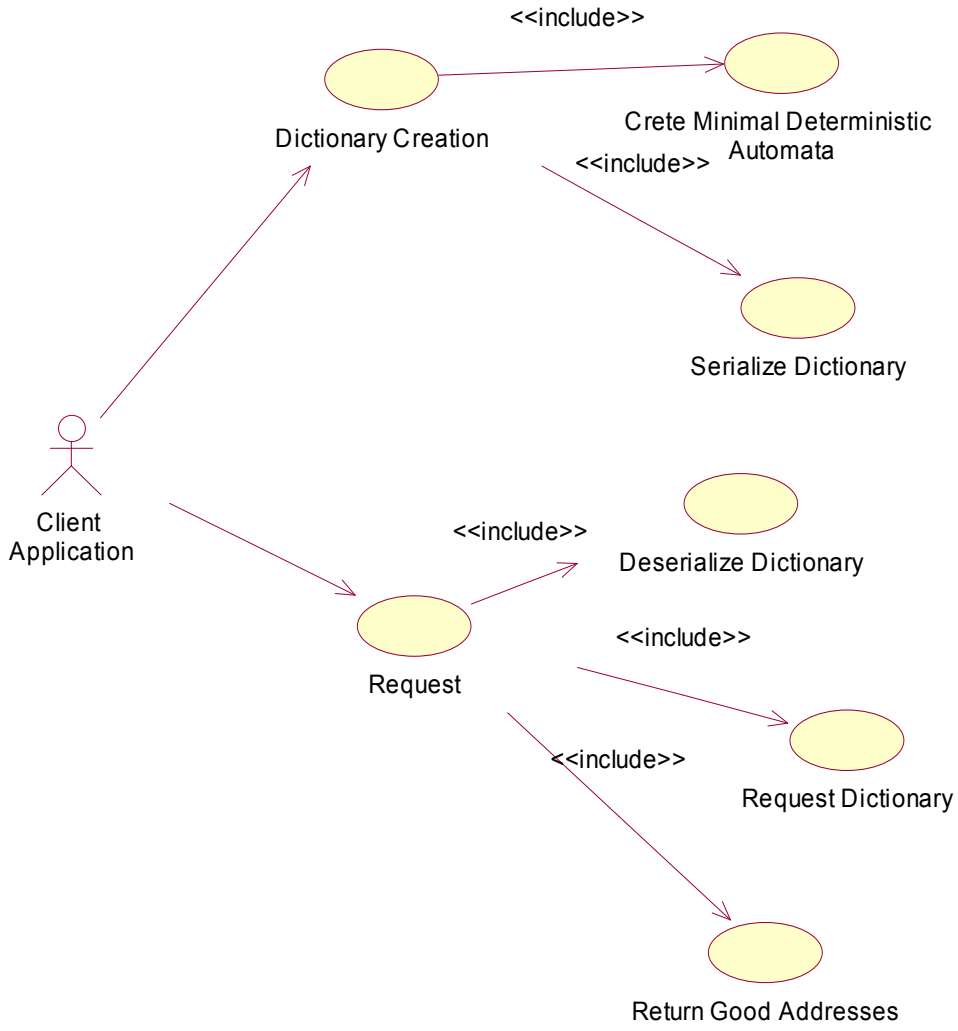
4.2 Модел

Проектираното решение трябва да бъде гъвкаво и да може да се разширява с минимални усилия. Разширяемостта на решението разбира се не е напълно безгранична и не трябва да излиза извън границите на планираната функционалност и интересите на фирмата. В противен случай ще се излезе извън определената функционалност и цели на реализация, което само по себе си е ново задание за решение.

При планирането на реализацията е избрано представяне, което изисква минимални корекции при добавяне на нова функционалност, за да бъде решението по-гъвкаво и системните изисквания да са минимални. За да бъде оформено решението във вида, в който то ще бъде представено в програмната му реализация в следващата секция, системата е моделирана с UML. Представеният UML модел преминава през няколко фази на моделиране на една система.

На първо място по ред на проектиране стои диаграмата на потребителските случаи – Use Case Diagram. Потребителят (в случая клиентското приложение) трябва да може да изпълнява две неща: да създава речник от списък с еднородни географски

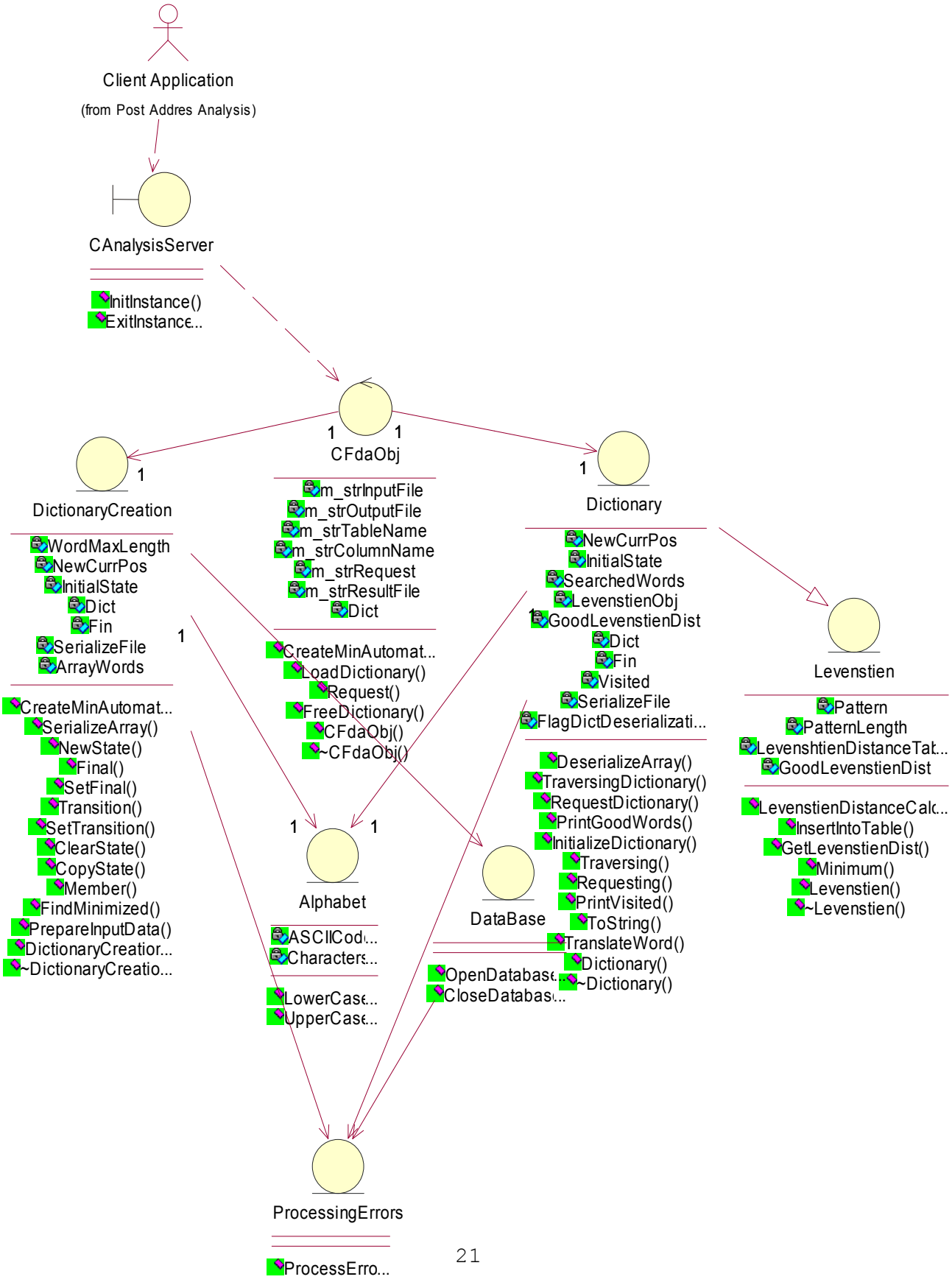
обекти и да прави запитвания за съществуването на адрес или негово приближение. Първият потребителски случай се извършва еднократно, когато базата се променя, а вторият използва построеният от първия речник при всяко свое запитване.

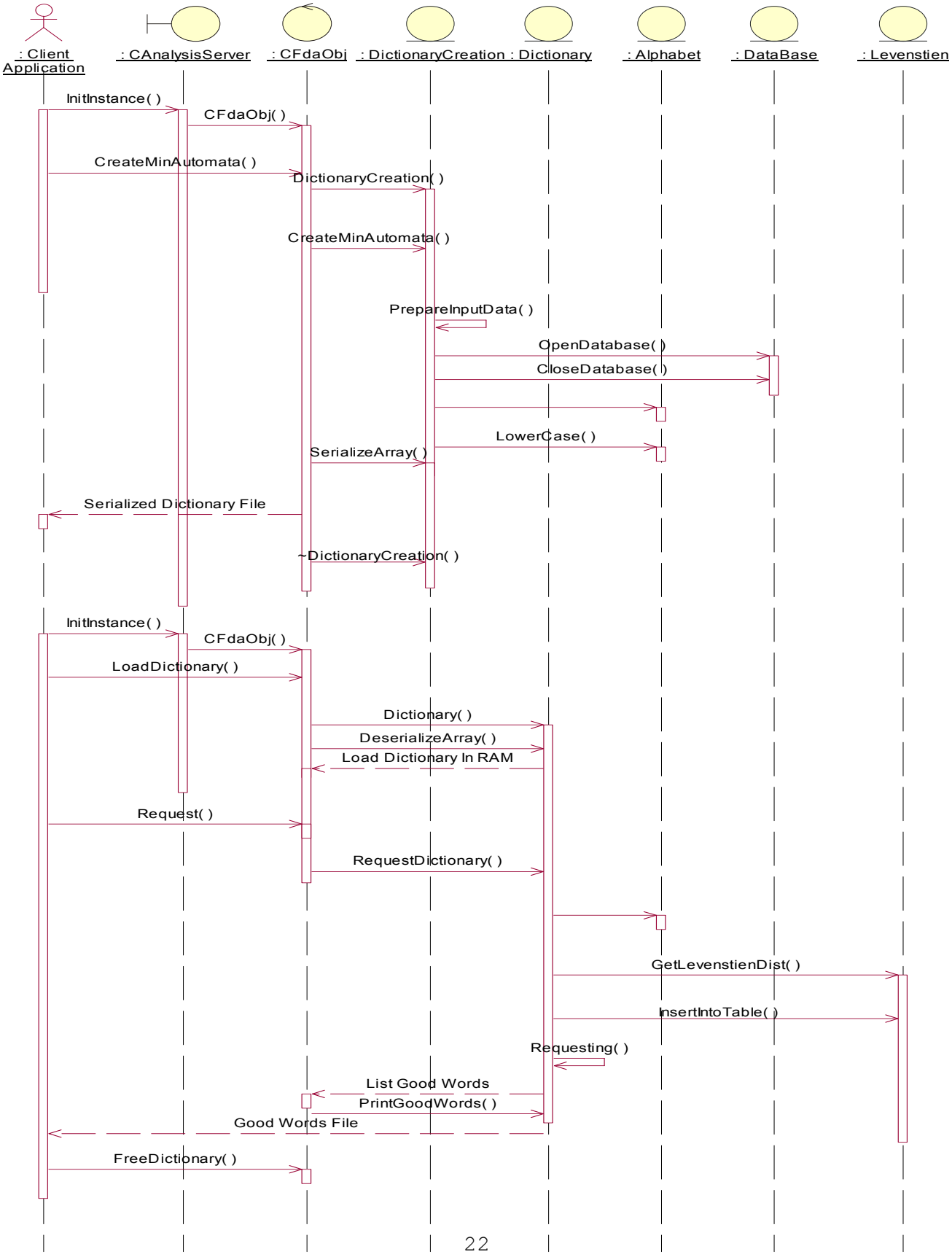


Диаграма 1. Диаграма на потребителските случаи

Следва моделирането на клас диаграмата – Class Diagram. Тя помества всички класове, който трябва да бъдат имплементирани с описание на функционалните и информационните им полета, както и връзките и взаимодействията между тях.

Диаграма 2. Клас диаграма

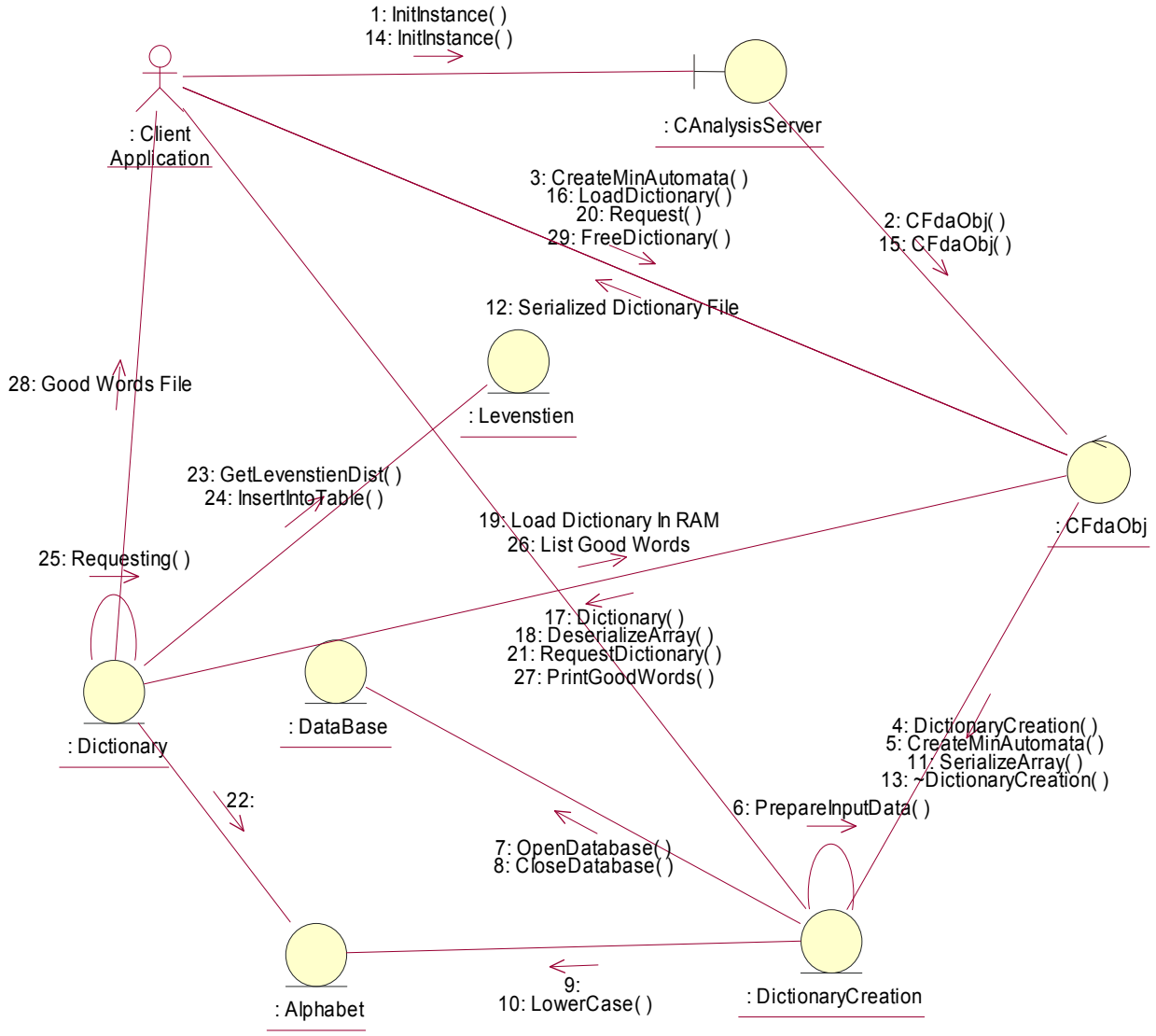

Client Application
(from Post Addres Analysis)
CAnalysisServer
InitInstance()
ExitInstance...
1
1
CFdaObj
m_strInputFile
m_strOutputFile
m_strTableName
m_strColumnName
m_strRequest
m_strResultFile
Dict
CreateMinAutomat...
LoadDictionary()
Request()
FreeDictionary()
CFdaObj()
~CFdaObj()
1
DictionaryCreation
WordMaxLength
NewCurrPos
InitialState
Dict
Fin
SerializeFile
ArrayWords
CreateMinAutomat...
SerializeArray()
NewState()
Final()
SetFinal()
Transition()
SetTransition()
ClearState()
CopyState()
Member()
FindMinimized()
PrepareInputData()
DictionaryCreatior...
~DictionaryCreatio...
1
Dictionary
NewCurrPos
InitialState
SearchedWords
LevenstienObj
GoodLevenstienDist
Dict
Fin
Visited
SerializeFile
FlagDictDeserializati...
DeserializeArray()
TraversingDictionary()
RequestDictionary()
PrintGoodWords()
InitializeDictionary()
Traversing()
Requesting()
PrintVisited()
ToString()
TranslateWord()
Dictionary()
~Dictionary()
Levenstien
Pattern
PatternLength
LevenshtienDistanceTat...
GoodLevenstienDist
LevenstienDistanceCalc...
InsertIntoTable()
GetLevenstienDist()
Minimum()
Levenstien()
~Levenstien()
1
1
Alphabet
ASCIICod...
Characters...
LowerCase...
UpperCase...
DataBase
OpenDatabase...
CloseDatabas...
ProcessingErrors
ProcessErro...


: Client Application
: CAnalysisServer
: CFdaObj
: DictionaryCreation
: Dictionary
: Alphabet
: DataBase
: Levenstien
InitInstance( )
CFdaObj( )
CreateMinAutomata( )
DictionaryCreation( )
CreateMinAutomata( )
PrepareInputData( )
OpenDatabase( )
CloseDatabase( )
LowerCase( )
SerializeArray( )
Serialized Dictionary File
~DictionaryCreation( )
InitInstance( )
CFdaObj( )
LoadDictionary( )
Dictionary( )
DeserializeArray( )
Load Dictionary In RAM
Request( )
RequestDictionary( )
GetLevenstienDist( )
InsertIntoTable( )
Requesting( )
List Good Words
PrintGoodWords( )
Good Words File
FreeDictionary( )

Диаграма 3. Диаграма на последователностите

Диаграмата на последователностите (Sequence Diagram) дефинира обръщенията към методите на класовете и в каква последователност те се извършват, за да бъде постигната желаната функционалност.

Диаграмата на съдействието (Collaboration Diagram) моделира точно по-какъв начин и в каква последователност във времето си взаимодействат различните инстанции на обектите. Тя зависи от диаграмата на последователностите и се генерира от нея.



Диаграма 4. Диаграма на съдействието

5 Софтуерна реализация

Поради естеството на задачата, която е основа на дипломната работа и при анализиране на възможностите, които трябва да има програмната система, става ясно че клиент-сървър моделът е най-подходящата архитектура при програмирането на системата. Това, което по същество е необходимо, е процес, който да бъде пуснат на някакъв хост и да обменя (приема и предава) информация с клиент, който му праща запитвания.

Сървърът е софтуерен пакет, осигуряващ специален вид услуги на клиентския софтуер. В клиент-сървър програмният модел, сървърът е програмата, която очаква(изчаква) и изпълнява запитвания от клиентските програми на същия или друг компютър. Обикновено той е разделен на файлово обслужване, позволяващо потребителите да съхраняват и достъпват информация на компютъра, и приложно обслужване, където софтуера подкарва програма, която изпълнява задачите на потребителите.

Разглежданата програмна система е реализирана като сървър приложение, предоставящо интерфейсни функции, с които клиентите да го достъпват. За да бъдат демонстрирани възможностите на сървъра е реализирано просто клиентско приложение.

5.1 Сървър

Сървърът е изграден на базата на два основни момента: създаването на минимален краен автомат (и съхраняването му на твърд носител) и претърсването му когато е получено запитване за някакъв адрес (което от своя страна включва предварителното зареждане на автомата в RAM паметта на компютъра еднократно).

На базата на тези функционални изисквания е изграден и интерфейса на COM обекта, какъвто имплементираната програмната система действително представлява.

5.1.1 Директно построяване на минимален ацикличен краен автомат по даден списък

Решаването на задачата за създаване на минимален автомат е важна за приложението на крайните автомати. До момента за построяване на минимален автомат, разпознаващ даден списък от думи, най-ефективен е методът на Revuz. При този метод се строи първо детерминиран краен автомат с дървовидна структура, който след това се минимизира.

Крайният детерминиран автомат е абстрактна математическа машина.[4] Всяка такава машина може да се разглежда като “черна кутия”, която чете от входа си думи над дадена азбука и може да извежда на изхода думи над друга (не непременно различна от входната) азбука. Характерно е, че във всеки момент от работата си машината се намира в някое състояние,

принадлежащо на крайно множество от състояния. Смяната на едно състояние с друго става в дискретно (изброимо) множество от моменти на времето, наричани тактове. Между всеки два такта машината остава в едно и също състояние. Новото състояние се определя еднозначно от текущото състояние и входната буква, която машината чете в тактовия момент. Изходната буква, когато машината действително извежда нещо, също е функция на текущото състояние и входната буква.

В началото на работата си абстрактната математическа машина винаги се намира в едно и също състояние, наричано начално състояние и работата и се определя от цикличното повтаряне на няколко прости действия – прочитане на входна буква, определяне от тази буква и текущото състояние на изходна буква и следващо състояние, извеждане на изходната буква и смяна на текущото състояние със следващото, изчакване до настъпване на следващия такт, когато действията се повтарят отново.

Абстрактната машина може да завърши работата, когато достигне някое от предварително фиксираните заключителни състояния или когато функцията, определяща следващото състояние е не навсякъде дефинирана (машината не може да продължи работата заради тази недефинираност). Разбира се, спирането може да се извърши и при прочитане на входната дума докрай или при някое друго, свързано с конкретната дефиниция събитие.

Дефиниция: Краен детерминиран автомат наричаме петорката $A=\langle \Sigma, S, s, F, \mu \rangle$, в която: Σ е крайна входна азбука; S е крайно множество от състояния; $s \in S$ е начално състояние; $\mu : S \times \Sigma \to S$ е частична функция на преходите, пресмятаща следващото състояние; $F \subseteq S$ са заключителните състояния на крайния детерминиран автомат.

Трябва да се отбележи, че разглежданите автомати не произвеждат изход. Те са частен случай на по-общото понятие абстрактна математическа машина, която включва изходна азбука и функция пресмятаща изходите.

Крайният детерминиран автомат се нарича минимален, когато за всеки друг автомат, разпознаващ същия език, имаме че броят на състоянията на минималния е по-малък или равен от броя на състоянията на останалите автомати.

Това, което ни интересува за конкретната реализация, е построяването на минимален краен детерминиран автомат, разпознаващ езика, състоящ се от всички думи в зададения списък от базата данни и само от тях.

Построяването на минимален ацикличен краен автомат по даден списък е осъществено по алгоритъма на доктор Стоян Михов, чиято дисертация в Българската академия на науките е била на

тема "Минимален ацикличен автомат: Конструиране, Алгоритми, Приложения". [1]

В неговата дисертация е въведено понятието минимален освен в дадена дума автомат. Това понятие е основната математическа концепция за представянията на метода за директно построяване на минимален автомат по даден списък, който ще бъде реализиран.

Нека имаме крайна азбука Σ с линейна наредба. Под наредба на думите в Σ^* се разбира лексикографската наредба, индуцирана от линейната наредба в Σ.

Дефиниция: Нека $A=\langle \Sigma, S, s, F, \mu \rangle$ е ацикличен детерминиран краен автомат с език $L(A)$. Тогава автомата A ще наричаме минимален освен в дадена дума $\omega \in \Sigma^*$, когато са изпълнени следните условия:

- Всяко състояние на автомата е достижимо от началното и от всяко състояние е достижимо крайно;
- ω е префикс на последната дума в лексикографската подредба на $L(A)$; В този случай можем да въведем следната нотация:

  $\omega = \omega_1^A \omega_2^A ... \omega_k^A$, където $\omega_i^A \in \Sigma$ за $i = 1,2,...,k$

  $t_0^A = s$; $t_1^A = \mu(t_0^A, \omega_1^A)$; $t_2^A = \mu(t_1^A, \omega_2^A)$; . . . ; $t_k^A = \mu(t_{k-1}^A, \omega_k^A)$;

  $T = \{t_0^A, t_1^A, ..., t_k^A\}$
- В множеството $S \setminus T$ няма различни еквивалентни състояния.
- $\forall r \in S \forall i \in \{1,2,...k\} \forall a \in \Sigma (\mu(r,a) \cong t_i \leftrightarrow (i < 0 \,\&\, r = t_{i-1} \,\&\, a = \omega_i^A))$.

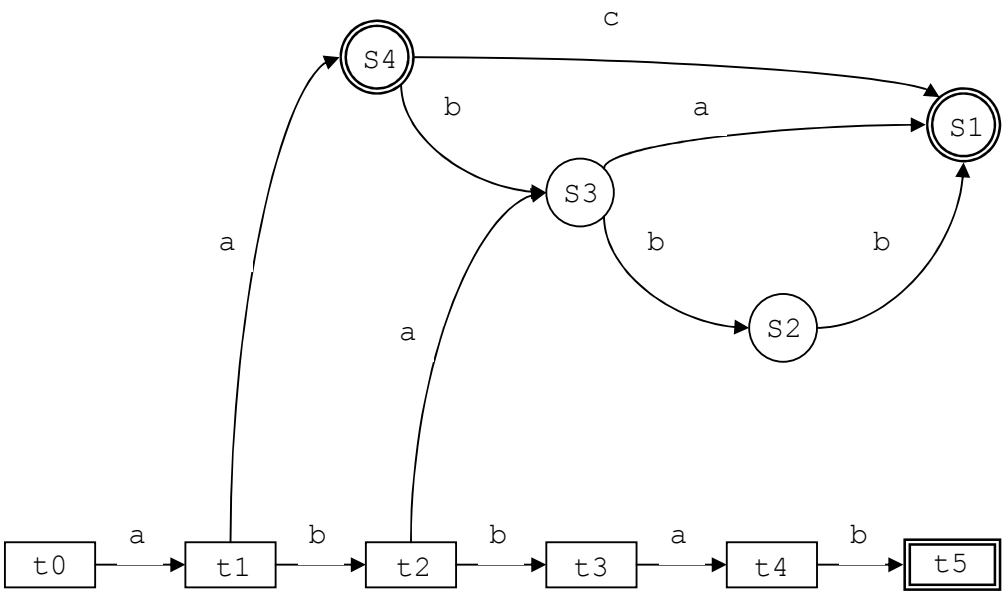
От тук нататък в случай, че това не води до двусмислие ще пишем t_i, ω_i вместо t_i^A, ω_i^A.

Ясно е, че ако автомат е минимален освен в две различни думи, то едната е префикс на другата.

Пример 1: На фигура 5 е даден ацикличен краен автомат над азбуката $\{a,b,c\}$. Езикът на автомата е $\{aa, aac, aaba, aabbb, abaa, ababb, abbab\}$. Този автомат е минимален освен в $abbab$ (при използване на обичайната наредба на азбуката).

Свойство: Ако един автомат е минимален освен в празната дума ε, то той е минимален.

Лема 1: Нека автоматът $A=\langle \Sigma, S, s, F, \mu \rangle$ е минимален освен в думата $\omega = \omega_1 \omega_2 ... \omega_k, \omega \neq \varepsilon$. Нека няма състояние еквивалентно на t_k в множеството $S \setminus T$. Тогава A е минимален също освен в думата $\omega' = \omega_1 \omega_2 ... \omega_{k-1}$.



Фигура 5. Краен автомат минимален освен в abbab

Лема 2: Нека автоматът $A=\langle\Sigma,S,s,F,\mu\rangle$ е минимален освен в думата $\omega=\omega_1\omega_2...\omega_k,\omega\neq\varepsilon$. Нека състоянието $p\in S\setminus T$ е еквивалентно на състоянието t_k. Тогава автоматът $A'=\langle\Sigma,S',s,F',\mu'\rangle$, дефиниран както следва:

$$S'=S\setminus\{t_k\}$$

$$F'=F\setminus\{t_k\}$$

$$\mu'(r,a)=\begin{cases}\mu(r,a), & \text{в случай, че } r\neq t_{k-1}\vee a\neq\omega_k \text{ и } !\mu(r,a)\\ p, & \text{вслучай, че } r=t_{k-1}, a=\omega_k,\\ \text{не е дефинирано} & \text{в противен случай}\end{cases}$$

е еквивалентен на автомата A и е минимален освен в $\omega'=\omega_1\omega_2...\omega_{k-1}$.

Теорема: Нека автоматът $A=\langle\Sigma,S,s,F,\mu\rangle$ е минимален освен в $\omega'=\omega_1\omega_2...\omega_m$. Нека $\psi\in L(A)$ е последната дума в лексикографската подредба на езика на автомата. Нека $\omega\in\Sigma^*$ е дума, която е по-голяма по лексикографски ред от думата ψ. Нека ω' е най-дългия общ префикс на думите ω и ψ. В този случай да означим $\omega=\omega_1\omega_2...\omega_m\omega_{m+1}\omega_k; k>m$. Тогава автоматът $A'=\langle\Sigma,S',s,F',\mu'\rangle$, дефиниран както следва:

$t_{m+1},t_{m+2},...,t_k$ са нови състояния такива, че $S\bigcap\{t_{m+1},t_{m+2},...,t_k\}=\varnothing$

$$S'=S\bigcup\{t_{m+1},t_{m+2},...,t_k\}$$

$$F'=F\bigcup\{t_k\}$$

$$\mu'(r,a)=\begin{cases} t_{i+1}, & \textit{в случай, че } r=t_i, m\le i\le k-1, a=\omega_{i+1} \\ \mu(r,a), & \textit{в случай, че } r\in Su!\,\mu(r,a) \textit{ и } p\ne t_m i\vee a\ne\omega_{m+1}, \\ \textit{не е дефинирано} & \textit{в противен случай} \end{cases}$$

е минимален освен в ω и разпознава езика $L(A)\bigcup\{\omega\}$.

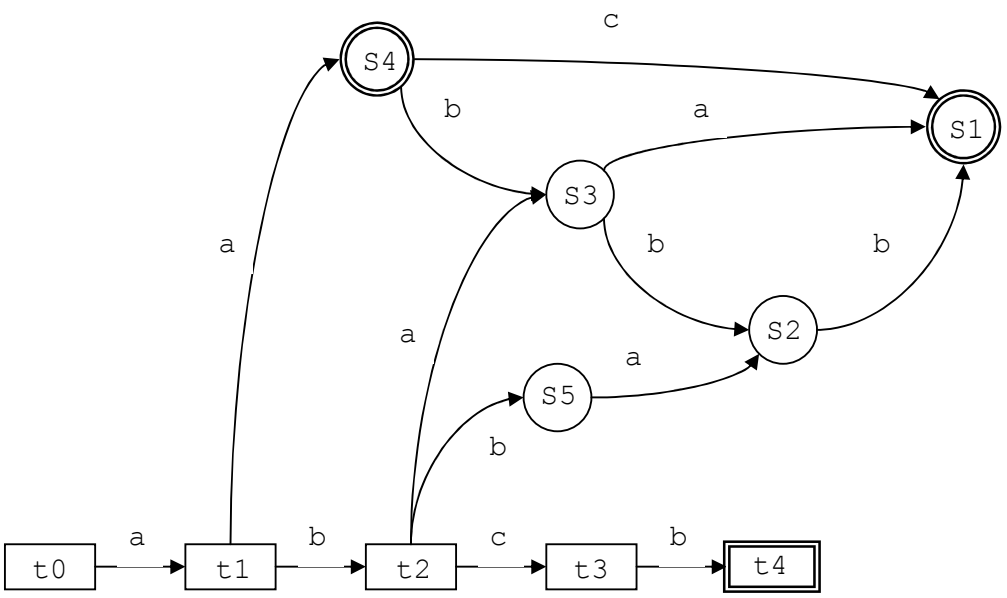
Тук се използва един директен метод за построяване на краен детерминиран автомат, който се основава на изложените по-горе дефиниция, леми и теорема и строи директно минималния автомат. Представящ дадения списък. Ще илюстрираме действието му с един пример.

Нека е даден непразен списък от думи L в лексикографска подредба. Нека с $\omega^{(i)}$ означим i-тата дума от списъка. Ще започнем с минималния автомат, разпознаващ точно първата дума от списъка. Този автомат се строи тривиално и е също минимален освен в $\omega^{(i)}$. Използвайки го за база провеждаме индукция по думите в списъка. Нека предположим, че автоматът $A^{(n)}=\langle\Sigma,S,s,F,\mu\rangle$ с език $L^{(n)}=\{\omega^{(i)}\mid i=1,2,...,n\}$ е построен и че $A^{(n)}$ е минимален освен в ω'. Трябва да построим автомат $A^{(n+1)}$ с език $L^{(n+1)}=\{\omega^{(i)}\mid i=1,2,...,n+1\}$, който е минимален освен в $\omega^{(n+1)}$.

Нека ω' е най-дългия общ префикс на думите $\omega^{(n)}$ и $\omega^{(n+1)}$. Използвайки няколко пъти лема 1 и лема 2 (в съответствие със случая) построяваме автомата $A'=\langle\Sigma,S',s,F',\mu'\rangle$, който е еквивалентен на $A^{(n)}$ и е минимален освен в ω'. Сека можем да ползваме теоремата и да построим автомата $A^{(n+1)}$ с език $L^{(n+1)}=L^{(n)}\bigcup\{\omega^{(n+1)}\}\bigcup\{\omega^{(i)}\mid i=1,2,...,n+1\}$, който е минимален освен в $\omega^{(n+1)}$.

По този начин индуктивно построяваме минималния освен в последната дума от списъка автомат, които разпознава списъка L. Накрая отново, използвайки неколкократно лема 1 и лема 2, построяваме еквивалентен автомат минимален освен в ε. От свойството имаме, че той е минималният автомат за списъка L.

При построяването на автомата броят на състоянията на междинния автомат е не по-голям от броя на състоянията на резултатния минимален автомат плюс дължината на най-дългата дума в списъка. Това е най-същественото предимство на нашия метод спрямо прилаганите досега.



Фигура 6. Краен автомат минимален освен в abcb

Пример 2: За да бъде илюстриран разглеждания метод, ще се спрем на следния пример. На фигура 6 е даден автоматът, разпознаващ $\{aa, aac, aaba, aabbb, abaa, ababb, abbab\}$. Този автомат е минимален освен в последната дума от списъка - $abbab$. Нека следващата дума е $abcb$. Най-дългият общ префикс на двете думи е ab. Трябва първо да построим автомат, еквивалентен на този, показан на фигура 5, който е минимален освен в ab като използваме лема 1 и лема 2. Първо прилагаме два пъти лема 2, след това веднъж лема 1. Накрая използваме теоремата и конструираме автомата минимален освен в $abcb$ даден на фигура 2. По този начин добавихме следващата дума от списъка към езика на автомата.

Алгоритъмът за директно построяване на минимален автомат за даден списък изисква абстрактни типове данни, представящи състояние на автомат и речник на състоянията. За представяне на речника на състоянията на автомата в конкретната реализация е избран двумерен масив – първият индекс на масива се мени по броя на състоянията, а втория индекс се мени по броя на буквите в дефинираната азбука. Тъй като броя на състоянията на автомата не е известен предварително и се определя(конструира) по време на изпълнение на програмата, един подходящ избор за имплементация на двумерен масив е контейнерът vector от стандартната библиотека на C++.

Контейнерът е обект, който съдържа други обекти. Такива са списъците, векторите и асоциативните масиви. Може да се добавят обекти към контейнер или да се отстраняват обекти от него. Контейнерите от стандартната библиотека на C++ са

проектирани така, че да отговарят на два критерия: да дават максимална свобода при създаването на отделния контейнер и едновременно с това да представят на потребителите единен интерфейс. Това дава възможност реализирането на контейнерите да бъде оптимално ефективно, а потребителския код да не зависи от това кой контейнер е използван. Контейнерите и алгоритмите (тази част от стандартната библиотека често е наричана STL) са решение, което е едновременно общо и ефективно. Тъй като средствата, предлагани от стандартната библиотека на C++, са предназначени да бъдат удобни, ефикасни и сравнително сигурни за обща употреба, STL-ият vector е един сигурен избор за реализация на динамичен двумерен масив. Един важен аспект на vector, който определи именно неговата употреба, в сравнение с другите контейнери е, че имаме лесен и ефективен достъп до отделните елементи в произволен ред.

При конкретният избор на реализация на речника на автомата състояние на автомата представлява индекс по редовете на двумерния масив. За имплементацията на данните и алгоритъма по построението на автомата е дефиниран класа DictionaryCreation:

```
class DictionaryCreation
{
public:
      // Constructorc and destructor
      .........
      .........

      void CreateMinAutomata(char* InFile, char* Table, char* Column);
      void SerializeArray();
private:
      int NewState();
      int Final(int State);
      void SetFinal(int State, int IsFinal);
      int Transition(int s, char c);
      void SetTransition(int r, char c, int s);
      void ClearState(int s);
      int CopyState(int s);
      int Member(int State);
      int FindMinimized(int s);
      int PrepareInputData(char* InFile, char* Table, char* Column);

      int WordMaxLength;           // max posiible length of word
      int NewCurrPos;              // states count; point new possible
                                   // position in Dict

      int InitialState;            // Initial state in dictionary

      List Dict;                   // Dictionary of states and transitions
      Array Fin;                   // Final states indicator array

      char* SerializeFile;         // Serializating file name
      ListWords ArrayWords;        // Contain sort words from input file
};
```

Член-данната NewCurrPos съдържа индекса на нова свободна позиция в масива, която може да помести ново състояние(постоянно или временно) на автомата. WordMaxLength

съдържа максималната възможна дължина на дума от речника. Важна роля за по нататъшното използване на получения автомат заема данната InitialState, която пази индекса на началното състояние на новопостроения автомат, който извежда всички думи от зададената база и само тях.

Dict съдържа таблицата с преходите в автомата. Реализиран е като vector от vector-и с дължина броя на буквите в азбуката на речника. Fin е vector, който съдържа информация за статуса на състоянията на автомата, разпознаващ езика на базата – крайно състояние или междинно състояние.

Върху автоматното състояние ще са ни необходими следните типове и операции:

- State е тип указател(в нашия случай индекс) към състояние на автомат;
- Буквите от азбуката на автомата са поредни и подредени по лексикографски ред – това е един от ключовите моменти в алгоритъма на Михов;
- Член-функция int NewState() връща ново празно състояние;
- Член-функция int Final(int State) връща истина, ако състоянието е крайно, и лъжа в противен случай;
- Член-функцията void SetFinal(int State, int IsFinal) задава крайност на състоянието;
- Член-функцията int Transition(int State, char Ch) връща състоянието, към което се преминава при преход, от състоянието, дадено като първи параметър чрез параметъра буква;
- Член-функцията void SetTransition(int State1, char Ch, int State2) насочва прехода от състоянието, дадено като първи параметър, чрез параметъра буква към състоянието, дадено като последен параметър;
- Процедура void PrintAutomata() извежда автомата с начало състоянието, дадено като параметър на файл.

Забележка: В математическата дефиниция състоянието е елемент от множеството на състоянията за даден автомат. В модела, който сега разглеждаме, състоянието само за себе си определя дали е крайно, както и функцията на преходите върху него. Имайки състоянието – член на абстрактен тип данни, ние можем да определим с точност до изоморфизъм подавтомата с начало това състояние. Тук използваме т.н. “автоматно състояние”, което дава възможност при предаване на дадено състояние като аргумент на дадена процедура да избегнем предаването като допълнителен аргумент на самия автомат.

Използвайки горните операции можем да дефинираме следните помощни член-функции:

- int CopyState(int State) копира състояние:

```
int DictionaryCreation::CopyState(int State)
{
   int NewSt;
```

```
   NewSt = NewState();
   SetFinal(NewSt, Final(State));

   // copy all transition of State in NewSt
   for (int I = 0; I < CharMapL; I++)
   {
      SetTransition(NewSt,Characters[I],Transition(State,Characters[]));
   }
   return NewSt;
}
```

- void ClearState(int State) изчиства състояние(анулира всички преходи от това състояние към други състояния):

```
void DictionaryCreation::ClearState(int State)
{
   SetFinal(State, 0);

   // delete all transition from State
   for (int I = 0; I < CharMapL; I++)
   {
      SetTransition(State, Characters[I], MissingTransition);
   }
}
```

- int Member(int State) проверява дали дадено състояние присъства вече в дотук построения речник:

```
int DictionaryCreation::Member(int State)
{
   // index of the permanent states
   int PermanentState = WordMaxLength + 1;

   // loop trough all permanent states
   while(PermanentState < NewCurrPos)
   {
      // loop trough all characters
      for (int LetterInd = 0; LetterInd < CharMapL; LetterInd++)
      {
         // if there is different transitions, break the loop trough the
         // other characters
         // for this state
         if(Dict[PermanentState][LetterInd] != Dict[State][LetterInd])
         {
            break;
         }
      }
   // whether the states have same transitions and whether the states
   // have same final statuses
      // if it is true then PermanentState is searched state
      if((LetterInd == CharMapL) && (Fin[PermanentState] == Fin[State]))
      {
         return PermanentState;
      }
      PermanentState++;
   }
   return NULL;
}
```

Всички операции, включително CopyState(int State) и ClearState(int State) се изпълняват за константно време при фиксирана азбука на автомата.

Азбуката на автомата е записана в константния масив char Characters[CharMapL]. Там са и трябва да бъдат изредени всички символи, от които се състоят думите във входният списък. Тъй като вторият индекс в двумерния масив, където ще се съхраняват всички преходи на автомата, е по буквите на азбуката (т.е. по техният брой), те трябва да имат някаква дефинирана подредба, която да обозначава смисъла на всеки индекс. В общия случай буквите на нашата азбука не са всичките символи в ASCII кодовата таблица, следователно буквите не са поредни и самите техни кодове не могат да изпълняват ролята на индекси. Ето защо е дефиниран константният масив int ASCIICodes[256], чиито стойности са както следва: ако символът с ASCII код k не принадлежи на азбуката на автомата, тогава елементът с индекс k в масива ASCIICodes[] е със стойност -1; ако символът с ASCII код k принадлежи на азбуката на автомата, тогава елементът с индекс k в масива ASCIICodes[] е със стойност номера на индекса от масива Characters, на която позиция се намира този символ. По този начин от ASCII кода на символа ние ще разбираме на кой индекс точно от двумерния масив с преходите на автомата да запишем прехода от дадено състояние чрез този символ. Така не е необходимо в речника да заделяме памет за всички ASCII символи, при което повечето от колоните на символите няма да съдържат никакви преходи, а само за тези букви, които ни интересуват.

Сега сме готови да представим кода на самия алгоритъм – член-функцията void CreateMinAutomata (char* InFile, char* Table, char* Column):

```
void DictionaryCreation::CreateMinAutomata(char* InFile, char* Table, char*
Column)
{
   string CurrentWord, PreviousWord = "";
   int PrefixLengthPlus1 = 0;

   // Preparation of input data for the algorithm
   int ReturnCode = PrepareInputData(InFile, Table, Column);

   if(ReturnCode)
   {
      // temp rows in dictionary
      int *TempStates = new int[WordMaxLength + 1];

      // initialize TempStates array
      for(int I = 0; I <= WordMaxLength; I++)
      {
         TempStates[I] = NewState();
      }

      // create min deterministic automata
      while(!(ArrayWords.empty()))
      {
         CurrentWord = ArrayWords.front();
         ArrayWords.pop_front();

         // calculate the length of the longest common prefix of
```

```
         // CurrentWord and PreviousWord
         I = 1;
         while((I <= CurrentWord.length()) && (I <= PreviousWord.length())
         && (PreviousWord[I-1] == CurrentWord[I-1]))
         {
            I++;
         }
         PrefixLengthPlus1 = I;

         // minimize the temp states of the sufix in PreviosWord
         for(I = PreviousWord.length(); I >= PrefixLengthPlus1; I--)
         {
            SetTransition(TempStates[I-1],               PreviousWord[I-1],
            FindMinimized(TempStates[I]));
         }

         // create the chain of new states for CurrentWord
         for(I = PrefixLengthPlus1; I <= CurrentWord.length(); I++)
         {
            ClearState(TempStates[I]);
            SetTransition(TempStates[I-1],                CurrentWord[I-1],
            TempStates[I]);
         }

         // make the last state for the word final
         SetFinal(TempStates[CurrentWord.length()], 1);

         PreviousWord = CurrentWord;
      }

      // minimize the states of last word
      for(I = CurrentWord.length(); I >= 1; I--)
      {
         SetTransition(TempStates[I-1],                   CurrentWord[I-1],
         FindMinimized(TempStates[I]));
      }

      // find initial state of dictionary
      InitialState = FindMinimized(TempStates[0]);

      delete [] TempStates;
   }
   else
   {
      string ErrorMessage = "Problem with input data";

      ProcessError(ErrorMessage);

      // mark execution error presentation
      FlagExecutionErrors = true;
   }
}
```

Член-функцията CreateMinAutomata използва помощната функция int FindMinimized(int State), която връща еквивалентно на състоянието State състояние от речника. Ако такова състояние не съществува, добавя към речника копие на това състояние и връща копието:

```
int DictionaryCreation::FindMinimized(int State)
{
   int StateMem;
```

```
   StateMem = Member(State);
   // if there is not equal state
   if (StateMem == NULL)
   {
      StateMem = CopyState(State);
   }
   return StateMem;
}
```

Трябва обаче да не забравяме, че алгоритъма се прилага върху сортиран списък от думи. Тъй като по изискване на фирмата възложител на задачата, която е основа на дипломната работа, данните се съхраняват в MDB база от данни и никъде изрично не е казано, че данните са сортирани, ние сме длъжни да осигурим коректни входни условия за алгоритъма. Ето защо още преди за започне процедурата по директното строене на автомата е необходимо да извлечем данните от базата и да ги сортираме във лексикографска наредба. Член-функцията PrepareInputData(char* InFile, char* TableName, char* ColumnName) намира максималния размер на дума от базата, за която ще строим речника и сортира списъка с думите от входния файл в нарастващ ред:

```
int DictionaryCreation::PrepareInputData(
                                        char* InDataBaseFile,
                                        char* TableName,
                                        char* ColumnName
                                        )
{
   // initialize success indicator
   int ReturnCode = 1;

   string Word;

   int TempLength = 0, PrefixLengthPlus1 = 0;

   // Open database and read data
   // name of db
   CDaoDatabase *Database;

   // initialize the database pointer
   Database = NULL;

   // close any open database
   CloseDatabase(&Database);

   int RetCode = OpenDatabase(&Database, InDataBaseFile, FALSE);
   if (1 == RetCode)
   {
      CDaoRecordset Recordset(Database);

      string Query = "";
      Query.append("SELECT ");
      Query.append(ColumnName);
      Query.append(" FROM ");
      Query.append(TableName);

      try
      {
          Recordset.Open(AFX_DAO_USE_DEFAULT_TYPE, Query.c_str());
      }
```

```
      catch (CDaoException *e)
      {
         // create a message to display
         string ErrorMessage = "Couldn't execute database query ->
Exception: \n";
         ErrorMessage.append(e->m_pErrorInfo->m_strDescription);

         // indicate fatal error
         ReturnCode = 0;

         // not rethrowing, so delete exception
         e->Delete();

         ProcessError(ErrorMessage);

         // mark execution error presentation
         FlagExecutionErrors = true;
      }
      catch (CMemoryException *e)
      {
         // indicate fatal error
         ReturnCode = 0;

         // not rethrowing, so delete exception
         e->Delete();

         ProcessError("Failed to open database -> Memory exception
         thrown.");

         // mark execution error presentation
         FlagExecutionErrors = true;
      }

      if(ReturnCode)
      {
         bool FlagAllDataReadable = 1;

         // Move through records and find max word size
         while(!Recordset.IsEOF())
         {
            COleVariant Var;

            Var = Recordset.GetFieldValue(0);
            if(Var.vt != VT_BSTR)
            {
               FlagAllDataReadable = 0;
            }
            else
            {
               Word = Var.pcVal;
               TempLength = Word.length();
               if(WordMaxLength < Word.length())
               {
                  WordMaxLength = TempLength;
               }

               // Convert cyrillic word to lowercase
               Word = LowerCase(Word);

               // fill array with all words from the input file
               ArrayWords.push_back(Word);
            }
```

```
            Recordset.MoveNext( );
         }

         // sort the array with the words
         ArrayWords.sort();

         if(FlagAllDataReadable == 0)
         {
            ProcessError("Field value is different from text.");
         }
      }
   }
   else
   {
      string ErrorMessage = "Couldn't open database file \"";
      ErrorMessage.append(InDataBaseFile);
      ErrorMessage.append("\"");

      ReturnCode = 0;

      ProcessError(ErrorMessage);

      // mark execution error presentation
      FlagExecutionErrors = true;
   }

   // close any open database
   CloseDatabase(&Database);

   return ReturnCode;
}
```

В тази член-функция съществени са три момента. На първо място стои въпросът със сортирането на голям по размер списък от думи. За да сортираме редица се нуждаем от начин за сравняване на елементи. Алгоритмите sort() от стандартната библиотека на C++ изискват итератори за произволен достъп, т.е. те работят най-добре върху vector-и и подобни контейнери. Стандартният list обаче не предоставя итератори с произволен достъп, затова той трябва да бъде сортиран чрез специализирани за list операции. Операцията sort() предоставена за сортиране на списък е стабилна, в смисъл че тя запазва относителния ред на елементите с еквивалентни стойности.

Вторият важен момент във функцията PreapreInputData() е свързан с четенето на данни от MDB база. Входните данни идват от дадена колона на конкретна MDB база.

MDB базата, с данните на която искаме да оперираме, ще отворим с *CDaoDatabase* обект. Един CDaoDatabase обект представя връзка до база от данни, чрез която могат да се достъпват данните от базата. MFC DAO класовете за бази от данни са различни от MFC класовете за бази от данни на ODBC. Класа CDaoDatabase дава интерфейс подобен на този на ODBC класа CDatabase. Разликата е, че CDatabase достъпва DBMS чрез Open Database Connectivity (ODBC) и ODBC драйвер за DBMS. CDaoDatabase достъпва данните

чрез Data Access Object (DAO). За отварянето на съществуваща база от данни се конструира обект от класа CDaoDatabase и се извиква неговата функция Open. Всяка от тези техники лепи DAO обекта към workspace-а на колекцията от бази данни и отваря връзка към данните. Когато се конструират CDaoRecordset, CdaoTableDef или CDaoQueryDef обекти за опериране с данни на свързана база от данни, трябва в конструктора за тези обекти да се постави указател към CDaoDatabase обекта. Когато работата по конекцията е приключена трябва да бъде извикана Close член-функцията на CDaoDatabase класа, която правилно да затвори конекцията и да разруши CDaoDatabase обекта.

Всички тези DAO изисквания са изпълнени във функциите, които манипулират отварянето и затварянето на дадената база от данни - int OpenDatabase(CDaoDatabase **ppDatabase, CString fileName, BOOL bReportNoOpen /* = TRUE */) и void CloseDatabase(CDaoDatabase **ppDatabase):

```
int OpenDatabase(CDaoDatabase **ppDatabase, CString fileName, BOOL
bReportNoOpen /* = TRUE */)
{
   // initialize success indicator
   int nReturnCode = 1;

   // close and delete if necessary
   if (*ppDatabase != NULL)
   {
      if ((*ppDatabase)->IsOpen())
            CloseDatabase(ppDatabase);
      delete *ppDatabase;
   }

   // construct new database
   *ppDatabase = new CDaoDatabase;

   // failed to allocate
   if (ppDatabase == NULL)
      return -1; // fatal error

   // now open the database object with error checking
   try
   {
      // When opening a database created with Access 2000 through
      // MFC DAO classes in Visual C++, you get the following error
      // message:
      // Unrecognized database format.
      // This error message occurs because the MFC DAO classes that ship
      // with Visual C++ 6.0 load DAO 3.5 (Dao350.dll) by default. DAO 3.5
      // uses Jet 3.5,
      // which can only open Jet 3.5 format (or earlier) databases.
      // Access 2000 creates Jet 4.0 format database files, which are
      // unrecognizable to Jet 3.5.
      // To successfully open an Access 2000 database using the MFC DAO
      // classes, you need to use
      // DAO 3.6 (Dao360.dll). DAO 3.6 uses Jet 4.0, which can open any
      // available Access database format.
      // For your application to use version 3.6 of DAO, you must update
      // the version of MFC at run time
      // to MFC version 6.01. To do this depends on whether you are
      // building the application
```

```
      // to use the MFC DLL or to build with the static libraries for MFC.
      // If you are linking with the MFC DLL, you can specify that you want
      // MFC
      // to use DAO 3.6 by inserting the following line of code before you
      // open an Access 2000 database:
      AfxGetModuleState()->m_dwVersion = 0x0601;
      (*ppDatabase)->Open(fileName);
   }
   catch (CDaoException *e)
   {
      // special case->couldn't find the file, so it may be because
      // user specified a new file to open
      if (e->m_pErrorInfo->m_lErrorCode == 3024)
      {
         if (bReportNoOpen)
         {
            // create a message to display
            string ErrorMessage = "Couldn't open database -> Exception: ";
            ErrorMessage.append(e->m_pErrorInfo->m_strDescription);

            ProcessError(ErrorMessage);
         }

         // indicate failure but not fatal
         nReturnCode = 0;
      }
      else // other type of DAO exception->always report
      {
         // create a message to display
         string ErrorMessage = "Couldn't open database -> Exception: ";
         ErrorMessage.append(e->m_pErrorInfo->m_strDescription);

         // indicate fatal error
         nReturnCode = -1;

         ProcessError(ErrorMessage);
      }

      // not rethrowing, so delete exception
      e->Delete();

      delete *ppDatabase;
      *ppDatabase = NULL;
   }
   catch (CMemoryException *e)
   {
      // output status
      ProcessError("Failed to open database->Memory exception thrown.");

      // not rethrowing, so delete exception
      e->Delete();

      delete *ppDatabase;
      *ppDatabase = NULL;

      // indicate fatal error
      nReturnCode = -1;
   }

   return nReturnCode;
}
```

```
void CloseDatabase(CDaoDatabase **ppDatabase)
{
   // only process if the database object exists
   if (*ppDatabase != NULL)
   {
      if ((*ppDatabase)->IsOpen())
      {
         (*ppDatabase)->Close();
      }
      // closing doesn't delete the object
      delete *ppDatabase;
      *ppDatabase = NULL;
   }
   AfxDaoTerm();
}
```

Третият съществен момент в подготовката за изпълнението на алгоритъма обхваща въпроса с големите (главните) и малките букви. По-логично е да се разглежда азбука или само от малки или само от големи букви, защото те имат еднакво семантично значение. Ето защо при директното създаването на речника като автомат от краен подреден списък с думи, входният списък за алгоритъма ще съдържа само думи написани с малките букви (само от такива букви трябва да се състои и самата азбука). Функцията,която конвертира думите в малки букви е string LowerCase(string Str):

```
string LowerCase(string Str)
{
      int StrLen = Str.length();
      string ResStr = "";
      int ICh;

      for(int I = 0; I < StrLen; I++)
      {
            ICh = (int)Str[I];
            if(
                  ((ICh >= -64) && (ICh <= -33)) ||
                  ((ICh >= 192) && (ICh <= 223)) ||
                  ((ICh >= 65) && (ICh <= 90))
              )
            {
                  ResStr.append(1, Str[I] + 32);
            }
            else
            {
                  ResStr.append(1, Str[I]);
            }
      }

      return ResStr;
}
```

Веднъж построен автоматът трябва да бъде съхранен на твърд дисков носител. Построяването на речника се прави само веднъж. В последствие той се зарежда в паметта и се използва при възникнало запитване за дума.

Процесът на сериализацията на автомата е важен, тъй като вида, в който автомата ще бъде десериализиран по-късно има съществено отношение към целта на поставената задачата, а именно претърсването на базата за заявен адрес (квартал, улица, жилищен комплекс) и намирането на близки на него при дефинирано разстояние за близост. Речникът, представен от двумерния масив Dict е сериализиран във вида, в който е построен – отново като двумерен масив.

Друга важна информация, която трябва да бъде налична при зареждането на масива, са финалните състояния, както и началното състояние на автомата. Тази информацията също се съхранява в сериализирания файл.

Методът, съхраняващ автомата във файл се нарича void SerializeArray():

```
void DictionaryCreation::SerializeArray()
{
   fstream FsOut = fstream(SerializeFile, ios::out | ios::trunc);

   if(FsOut.is_open())
   {
      // memorize states count
      FsOut << "States count: " << NewCurrPos - (WordMaxLength + 1) <<
      "\n";
      // memorize initial state
      FsOut << "Initial state: " << InitialState - (WordMaxLength + 1) <<
      "\n";
      // memorize alphabet symbols count
      FsOut << "Characters: " << CharMapL << "\n";

      // memorize final state
      FsOut << "Final states: ";
      for(int I = WordMaxLength + 1; I < NewCurrPos; I++)
      {
         if(Fin[I] == 1)
         {
            FsOut << I - (WordMaxLength + 1) << " ";
         }
      }

      FsOut << "\n-------------\n";

      for(I = WordMaxLength + 1; I < NewCurrPos; I++)
      {
         for(int J = 0; J < CharMapL; J++)
         {
            // if Dict[I][J] == MissingTransition then just print
            // MissingTransition value
            if(Dict[I][J] != MissingTransition)
            {
               FsOut << Dict[I][J]  - (WordMaxLength + 1) << " ";
            }
            else
            {
               FsOut << MissingTransition << " ";
            }
         }
```

```
            FsOut << "\n";
        }

        // new initialization of class data
        NewCurrPos = NewCurrPos - (WordMaxLength + 1);
        InitialState = InitialState - (WordMaxLength + 1);
        WordMaxLength = MissingTransition;

        FsOut.close();
    }
    else
    {
        string ErrorMessage = "Couldn't open output file to serialization:
        \"";
        ErrorMessage.append(SerializeFile);
        ErrorMessage.append("\"");

        ProcessError(ErrorMessage);

        // mark execution error presentation
        FlagExecutionErrors = true;
    }
}
```

След построяването на речник от пощенските адреси(улици, квартали и т.н.) във вид на минимален детерминиран автомат, пред нас се изправят въпросите за разумно бързо обхождане само на адекватна на заявката част от базата данни и едновременно с това задачата при обхождането да се колекционират близките (в смисъл на Левенщайн) до заявката записи от базата.

5.1.2 Обхождане и претърсване на автомата за близки до заявката (в смисъл на Левенщайн) думи

След като автомата е вече построен и сериализиран, към него могат да бъдат подавани заявки (запитвания) за присъствието на дума в базата от данни. Преди сървърът да е готов да анализира и удовлетворява потребителки заявки, той трябва да извърши някои подготвителни действия като десериализирането на автомата и зареждането му в RAM паметта. Това зареждане се прави еднократно преди сървъра да премине в състояние на готовност за удовлетворяване на потребителски заявки.

Въпросът със сериализацията и десериализацията е важен, защото от тях зависи по-нататъшния избор на структури от данни за манипулация на речника. Изборът може да бъде направен измежду следните два подхода: сериализационния механизъм, осигурен от Microsoft Foundation Class Library (MFC) или собствени методи за осъществяването на сериализацията и десериализацията.

MFC сериализационният механизъм позволява обекти да бъдат запазени между две изпълнения на програмата. “Сериализацията” е процес на писане в и четене от постоянна среда за съхранение, такава като дисков файл. MFC поддържа сериализацията в класа CObject. По този начин всички класове,

наследяващи Cobject, могат да използват предимствата на сериализационния протокол на CObject.

Основната идея на сериализацията е че един обект може да запази текущото си състояние, обикновено индицирано от стойността на неговите член-данни, за постоянно съхранение. По-късно обекта може да създаде отново чрез четене, или десериализация, състоянието си от мястото на съхранение. Сериализацията държи всички детайли на обектните указатели и връзки към обектите, които са използвани, когато обекта е бил сериализиран. Една ключова точка тук е че обекта сам е отговорен за четенето и записа на собственото си състояние. По този начин за класа, който ще бъде сериализиран, трябва да се имплементират сериализационни операции.

MFC използва обект от класа CArchive като връзка между обекта, който ще бъде сериализиран и носителя за съхранение. Този обект винаги е асоцииран с обект от класа CFile, от който той получава необходимата информация за сериализацията, включваща име на файла и дали исканата операция е четене или писане. Обекта, който извършва сериализационна операция може да използва CArchive без да зависи от природата на носителя за съхранение. Всеки CArchive обект използва предефинираните оператори за вмъкване (<<) и извеждане(измъкване) (>>), за да осъществи операциите за писане и четене.

Това означава, че MFC сериализционния метод би бил подходящ за случаите, в които е необходимо структурите от данни, които се сериализират и десериализират да са едни и същи, за да могат да се капсулират в клас, който да наследи CObject класа. При конструирането на речника във вид на минимален детерминиран автомат е използвана структура от данни вектор от вектори, тъй като предварително броят на състоянията на автомата не е известен и се решава по време на изпълнение. Употребата на векторите от стандартната библиотека на C++ в случая е подходяща, защото те ще решат проблемите по заделянето на необходимата памет. При десериализацията обаче нещата стоят по друг начин, защото вече броя на състоянията е известен и може динамично да бъде заделена памет за двумерен масив, при който директния достъп до елементите се извършва по бързо отколкото при векторите от стандартната библиотека. Ето защо при десериализацията е излишно да се използва контейнера вектор, след като може просто да бъде използван динамичен двумерен масив. Аналогично стои въпроса със структурата от данни, в която е съхранен статуса на финалните състояния – при строенето на автомата и сериализацията на финалните състояния е добре да се използва вектор, но при десериализацията вектор не е необходим, защото едномерния масива би свършил същата работа, но би улеснил и забързал достъпа до елементите.

След този анализ става ясно, че написването на собствени методи за сериализация и десериализация би бил едно приемливо решение.

Данните и методите по обхождането и претърсването на автомата са капсулирани в класа *Dictionary:*

```
class Dictionary
{
public:
   // Constructors and destructor
   .........
   .........

   void DeserializeArray();
   void TraversingDictionary();
   void RequestDictionary(string Request, int LevenstienDistance);
   void PrintGoodWords(char* ResFile);

   bool FlagExecutionErrors;
private:
   void InitializeDictionary();
   void Traversing(int State);
   void Requesting(int State, int Index,  int PositionString);
   void RequestLevenstien(int InitialSubstringState, int PositionString,
   int LevenstienDistance);
   void PrintVisited();
   string ToString(int MinFindLevenshtienDist);
   string TranslateWord(string T);
   int RequestAnswer();
   string FindRequest(string Request);
   string OperateAbbreviationInRequest(string Request);
   string RemoveLastIntervals(string Request);

   int NewCurrPos;                      // states count; point   new
   possible position in Dict
   int InitialState;                    // Initial state in  dictionary

   GoodSearchResults SearchedWords;

   LevenstienObjs LevenstienObj;        // Levenstien distance objects

   int (*Dict)[CharMapL];               // Dictionary of states and
                                        // transitions
   int *Fin;                            // Final states indicator array

   Word Visited;                        // Visited letters

   char* SerializeFile;                 // Serializating file name

   bool FlagDictDeserialization;        // Marker for deserialization
                                        // availability
   vector<VecInt> MarkersDisplay;       // Marker for finded good words in
                                        // some sequence in ditionary
   vector<VecInt> FindedDist;
};
```

Метода void DeserializeArray() десериализира речника от текстовият файл, в който е записан, като го зарежда в двумерния масив Dict. Избора на двумерен масив като структурата от данни, в която ще се съдържат преходите от

състояние в състояние, с първи индекс, който се мени по броя на състоянията и втори индекс - по броя на буквите, е най-вече предопределен от възможността за директен достъп до произволен елемент от масива. В този случай обаче двумерния масив не е реализиран като вектор, за да няма излишно забавяне при обхождането му. По-точно е да се каже, че речника е бил сериализиран като двумерен масив, за да може после да бъде десериализиран като такъв по най-лесния начин.

В масива Fin се зарежда статусът на състоянията - крайно или междинно, а в член-данната InitialState при десериализацията се записва индекса на началното състояние на автомата. NewCurrPos съдържа броя на състоянията в автомата. Ето и сорс-кода на метода DeserializeArray():

```
void Dictionary::DeserializeArray()
{
   fstream FsIn = fstream(SerializeFile, ios::in);

   string Line;
   int Pos = 0, StatesPos, InitialStatePos, CharNumbersPos, FinalStatesPos;
   int BegPos = 0;
   int I = 0, Value, FinStatesLine;
   string IntValue, StatesValue, InitialStateValue, CharNumbersValue;

   if(FsIn.is_open())
   {
      // marker
      getline(FsIn, Line);
      if(!Line.compare("Automata Serialization File"))
      {
         // find number of states
         getline(FsIn, Line);
         StatesPos = Line.find(':', 0);
         StatesValue = Line.substr(StatesPos + 2, Line.length());
         Value = atoi(StatesValue.c_str());
         // states count initialization (NewCurrPos data in class
         // Dictionary)
         NewCurrPos = Value;

         // find initial state index
         getline(FsIn, Line);
         InitialStatePos = Line.find(':', 0);
         InitialStateValue    =    Line.substr(InitialStatePos    +    2,
         Line.length());
         Value = atoi(InitialStateValue.c_str());
         // initial state initialization (InitialState data in class
         // Dictionary)
         InitialState = Value;

         // find number of characters in alphabet
         getline(FsIn, Line);
         CharNumbersPos = Line.find(':', 0);
         CharNumbersValue = Line.substr(CharNumbersPos + 2, Line.length());
         Value = atoi(CharNumbersValue.c_str());

         // initialize new empty Dict
         // and initialize final states as non final
         InitializeDictionary();
```

```
      // find final states
      getline(FsIn, Line);
      FinalStatesPos = Line.find(':', 0);
      FinalStatesPos = FinalStatesPos + 2;
      FinStatesLine = Line.length();
      while(FinalStatesPos != FinStatesLine)
      {
         Pos = Line.find(' ', FinalStatesPos);
         IntValue = Line.substr(FinalStatesPos, Pos - FinalStatesPos);

         // Value is index of final state
         Value = atoi(IntValue.c_str());
         Fin[Value] = 1;

         FinalStatesPos = Pos + 1;
      }

      // skip line "-----------"
      getline(FsIn, Line);

      // complete Dict with all transition in the min deterministic
      // automata
      I = 0;
      while(!(FsIn.eof()))
      {
         getline(FsIn, Line);

         // skip last new line in serialized file
         if(Line == "")
         {
            break;
         }

         for(int J = 0; J < CharMapL; J++)
         {
            Pos = Line.find(' ', BegPos);
            IntValue = Line.substr(BegPos, Pos - BegPos);
            // Value is transition in automata from state with index I
            // by means of character with index J
            Value = atoi(IntValue.c_str());
            Dict[I][J] = Value;

            BegPos = Pos + 1;
         }

         I++;
         BegPos = 0;
      }

   FlagDictDeserialization = true;
}
else
{
   string ErrorMessage = "Couldn't deserialize file ";
   ErrorMessage.append(SerializeFile);
   ErrorMessage.append(".  It  is  not  serialized  with  AnlysisServer
   module");

   ProcessError(ErrorMessage);
}

FsIn.close();
```

```
}
else
{
   string ErrorMessage = "Couldn't open serialized file ";
   ErrorMessage.append(SerializeFile);

   ProcessError(ErrorMessage);
}
```

След като речника вече е зареден в RAM паметта на компютъра, той може да бъде обхождан и съответно претърсван. Построеният речник представлява минимален краен детерминирам автомат. Той се представя като ориентиран граф, с върхове елементите от множеството от състоянията на автомата, в които два върха $q_i, q_j \in S$ са свързани с ребро надписано с $x \in \Sigma$, ако $\mu(q_i, x) = q_j$. Под обхождане на граф се разбира процедура, която систематически, по определени правила, посещава ("разглежда") всички върхове на графа. Най-популярни са два вида обхождане, които могат да се използват като алгоритмични схеми. Има и други обхождания, които нямат качествата на алгоритмични схеми, а са специфични алгоритми, но те няма да бъдат разгледани. [4]

Обхождане в ширина: Съществено за тази алгоритмична схема е понятието *ниво на обхождане*, което представлява подмножество на множеството от върховете на графа $G(V, E)$. В резултат на обхождането в ширина, V се разбива на нива $L_0, L_1, ..., L_k$ по следния начин (по-долу под "обхождане" на връх разбираме някакви специфични действия, зависещи от конкретната задача):

1) Избираме *начален връх* r на обхождането (началният връх може и да е зададен предварително). Образуваме $L_0 = \{r\}$ и нека $l = 0$. Обявяваме върха r за "обходен".

2) Ако $L_0 \cup L_1 \cup ... \cup L_l = V$, тогава обхождането е завършено. В противен случай преминаваме към 3.

3) Нека сме обходили върховете от нивата $L_0, L_1, ..., L_l$. Образуваме нивото L_{l+1} от всички необходени върхове $v_i \notin L_0 \cup L_1 \cup ... \cup L_l$, които са съседи на върхове от L_l. Обявяваме върховете от L_{l+1} за "обходени". Нека $l = l + 1$ и преминаваме към 2.

Обхождане в дълбочина: При тази схема основни са понятията *текущ връх* t и *баща* на върха t – $p(t)$. Началният връх на обхождането r и тук е зададен предварително или е определен произволно. Обхождането става по следния начин:

1) "Обхождаме" началния връх r. При това $t = r$, а $p(t)$ е неопределен.

2) Търсим необходен връх, който е съседен на текущия t:

a) ако има такъв връх v, тогава ("стъпка напред"): $p(v) = t$, $t = v$ и обявяваме v за "обходен". Преминаваме към 2.

b) ако няма такъв връх:

b.1) ако $t \neq r$, $p(t)$ е определен. Тогава $t = p(t)$ и преминаваме към 2 ("стъпка назад");

b.2) ако $t = r$ обхождането е завършило.

С други думи, схемата за обхождане в дълбочина ни предписва да опитваме стъпки напред (в дълбочина), докато това е възможно, а в случай на невъзможност, да направим стъпка назад и отново да опитаме стъпка напред. Затова тази алгоритмична техника често е наричана *обхождане с връщане*.

За да бъдат изведени всички думи от езика, който разпознава даден автомат, трябва да се направи пълно обхождане на графа, който представя автомата, и то в дълбочина. Необходимо е да се правят стъпки напред докато това е възможно, за да се прочете последователност от букви, която ако достигне финално състояние, формира дума, която автомата (речника) разпознава. Когато обхождане напред вече не е възможно, трябва да се направи стъпка назад и да се опита отново напред за прочитането на следваща дума и т.н.

Функционалността, която реализира алгоритъма за обхождане на речника в дълбочина, е застъпена в метода void Traversing(int State). Този метод не е използван директно в кода на програмата, но е използван за тестване коректността на автомата (с негова помощ беше тествано дали езика, който извежда автомата, съвпада с базата от данни, която е зададена в началния момент като входен параметър). На базата на този метод е извършено претърсването на автомата за дадена дума или за нейни приближения в метода void Dictionary::Requesting(int State, int Index, int PositionString). Той е рекурсивна функция, която има за цел да премине през почти всички състояния на автомата, за да може да отговори на въпроса дали съществуват приближения на дадена дума и ако да – кои са те. Почти всички в смисъл, че ако прочетена последователност от букви не задоволява изискванията ни за близка до дадена дума и е сигурно, че всяка добавена буква до тази последователност няма начин да подобри близостта със зададената дума, то не е необходимо да се продължава търсенето по този клон на графа и той може да бъде изоставен. Ето и сорс кода на разглежданата рекурсивна функция:

```
void Dictionary::Requesting(int State, int Index,  int PositionString)
{
   ASSERT (State > -1);
   ASSERT (Index > -1);
   ASSERT (PositionString < LevenstienObj.size());

   // if State is final
   // then try to output answer
   if(Fin[State])
   {
      if(PositionString == LevenstienObj.size() - 1)
      {
         int Distance = LevenstienObj[PositionString]-
         >GetLevenstienDist(Index - 1);
```

```
      if((Distance != -1) && (Distance <= LevenstienObj[PositionString]-
      >GoodLevenstienDist))
      {
         MarkersDisplay[PositionString].push_back(Index - 1);
         FindedDist[PositionString].push_back(Distance);
      }
   }

   // if for all requested words exist approximation words in the given
   sequence
   int MinLevenshtien = RequestAnswer();
   if(MinLevenshtien != -1)
   {
      SearchedWords.push_back(ToString(MinLevenshtien));
   }
}

// unmark current marker
if(MarkersDisplay[PositionString].back() == Index - 1)
{
   MarkersDisplay[PositionString].pop_back();
   FindedDist[PositionString].pop_back();
}

// for all transitions of State
for(int I = 0; I < CharMapL; I++)
{
   if(Dict[State][I] != MissingTransition)
   {
      // read letter
      Visited.push_back(Characters[I]);

      // if readed letter is ' '
      // do special initialization or search for next word from
      requested sequence
      if(Characters[I] == ' ')
      {
         int Distance = LevenstienObj[PositionString]-
         >GetLevenstienDist(Index - 1);

         if((Distance != -1) && (Distance <=
         LevenstienObj[PositionString]->GoodLevenstienDist))
         {
            MarkersDisplay[PositionString].push_back(Index - 1);
            FindedDist[PositionString].push_back(Distance);
         }

         // if current searched word satisfies the Levenstien distance
         // search in the rest of the dictionary for next searched word
         from the given sequence
         if((PositionString < LevenstienObj.size() - 1) && (Distance !=
         -1) && (Distance <= LevenstienObj[PositionString]-
         >GoodLevenstienDist))
         {
            Requesting(Dict[State][I], 1, PositionString + 1);
         }
         // move calculation of Levenstien distance to the index after""
         else
         {
            LevenstienObj[PositionString]->IndexFirstColumnArray++;
```

```
                LevenstienObj[PositionString]-
                >FirstColumns[LevenstienObj[PositionString]-
                >IndexFirstColumnArray] = Index;

                LevenstienObj[PositionString]-
                >InsertIntoTable(Characters[I], Index);
                Requesting(Dict[State][I], Index + 1, PositionString);

             }

          }
          // continue recursion
          else
          {
             LevenstienObj[PositionString]->InsertIntoTable(Characters[I],
             Index);
             Requesting(Dict[State][I], Index + 1, PositionString);
          }

          char Ch = Visited.back();
          Visited.pop_back();

          if(Ch == ' ')
          {
             if(MarkersDisplay[PositionString].back() == Index - 1)
             {
                MarkersDisplay[PositionString].pop_back();
                FindedDist[PositionString].pop_back();
             }

             int Distance = LevenstienObj[PositionString]-
             >GetLevenstienDist(Index - 1);
             if((PositionString < LevenstienObj.size() - 1) && (Distance !=
             -1) && (Distance <= LevenstienObj[PositionString]-
             >GoodLevenstienDist))
             {
             }
             // move calculation of Levenstien distance to the index
             before''
             else
             {
                LevenstienObj[PositionString]->IndexFirstColumnArray--;
                LevenstienObj[PositionString]-
                >InitializeInitial(LevenstienObj[PositionString]-
                >FirstColumns[LevenstienObj[PositionString]-
                >IndexFirstColumnArray]);
             }
          }
       }
    }
}
```

Идва момента за дефинирането на близост между две думи (между два адреса/части от адреси), което е мярката, по която горепредставената функция извежда подходящите приближения. Под близки думи в конкретния случай ще се разбира близост в смисъл на Левенщайн.

Пресмятането на Левенщайн разстоянието е един пример за динамично програмиране. Динамичното програмиране е техника за

построяване на структури от данни и алгоритми. Техниката динамично програмиране е използвана най-често за оптимизационни задачи, където се иска да се намери най-добрия начин да се направи нещо. Често броят на различните начини за направата на нещо расте с размера на нещото, така че търсенето "груба-сила" е най-добре да се прилага върху проблемите с най-малки размери. Техниката динамично програмиране се прилага в случаите, в които проблема има определена структура, която може да се използва (експлоатира).
Тази структура включва следните три компонента:

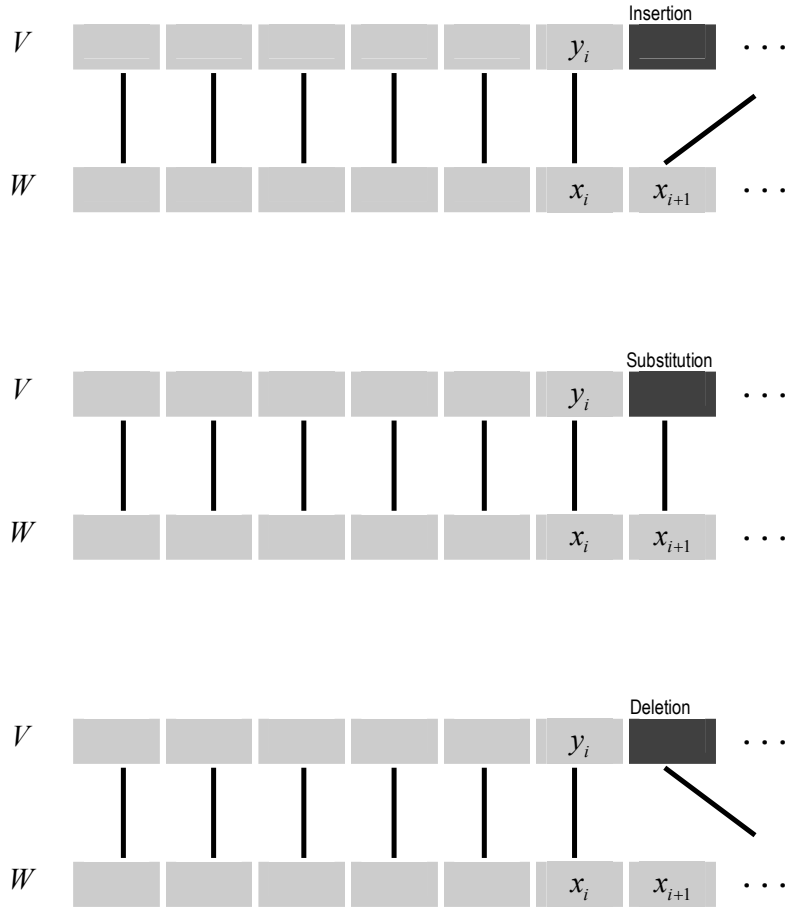
- Прости подпроблеми: Трябва да има начин за разделянето на големите оптимизационни проблеми на подпроблеми. Още повече, трябва да има прост начин за дефинирането на подпроблемите просто като индекси (като i, j, k и така нататък).
- Оптимизация на подпроблемите: Оптималното решение на глобалните проблеми трябва да бъде съвкупност от оптимални решения на подпроблемите. Ние не трябва да можем да намираме глобално оптимално решение, което съдържа неоптимални подпроблеми.
- Застъпване на подпроблемите: Оптималните решения на несвързани подпроблеми могат да съдържат едни и същи подпроблеми. [6]

Под разстояние на Левенщайн между две думи се разбира минималният брой вмъквания, зачеркквания или заместквания, които са необходими за преобразуването на едната дума в другата. Левенщайн разстоянието между две думи е базирано на идеята за примитивните редактиращи операции. Примитивни операции тук са заместването на символ с друг символ, зачеркването на символ, и вмъкването на символ. Очевидно, за всеки две думи V и W над азбука Σ винаги е възможно да пренапишем V в W, използвайки примитивните редактиращи операции. [5]

Дефиниция на разстояние на Левенщайн: Нека V и W са думи над азбуката Σ. Левенщайн разстояние между V и W е минималният брой редактиращи операции (заместквания, зачеркквания или вмъквания), които са необходими да преобразуват V в W. С $d_L(V,W)$ се означава Левенщайн разстоянието между V и W. То може да бъде пресметнато като се използва следната проста схема на динамичното програмиране:

$$d_L(\varepsilon, W) = |W|$$
$$d_L(V, \varepsilon) = |V|$$
$$d_L(aV, bW) = \begin{cases} d_L(V,W), & ако\ \ a = b \\ 1 + \min(d_L(V,W), d_L(aV,W), d_L(V,bW)), & ако\ \ a \neq b \end{cases}$$

за $V, W \in \Sigma^*$ и $a, b \in \Sigma$.



Фигура 7. Примитивни редактиращи операции за пренаписването на дума в друга

Както беше споменато по-горе един от ключовите компоненти в техниката динамично програмиране е дефинирането на прости подпроблеми, които да задоволят свойствата подпроблемна оптимизация и подпроблемно припокриване. От това че V и W са символни низове, ние имаме натурално множество от индекси, с които да дефинираме подпроблемите – индексите в стринговете V и W. Дефиницията на Левенщайн разстоянието ни позволява да дефинираме $d_L(V,W)$ в термините на оптимални решения на подпроблемите.

Горната дефиниция на $d_L(V,W)$ удовлетворява подпроблемната оптимизация, заради това че не е възможно да имаме разстояние на Левенщайн за дадени думи, ако нямаме разстояние на Левенщайн за техните поддуми. Също така тази дефиниция използва и припокриване на подпроблеми, защото едно подрпоблемно решение $d_L(V,W)$ може да бъде използвано в няколко други проблема(именно $d_L(aV,W)$, $d_L(V,bW)$ и $d_L(aV,bW)$).

Използвайки построената таблица може да се конструира оптимално подравняване (подравняването може да е повече от

едно). Оптималното подравняване/подравнявания са представени чрез пътя през матрицата, изобразен чрез употребата на друг алгоритъм – алгоритъм с връщане назад (traceback algorithm). Той изобразява предшественика на всяка клетка от пътя на оптималното подравняване, стартирайки от долния десен ъгъл. Намирането на предшественика на някоя клетка за определянето на оптималната следа на подравняване се идентифицира с:

a) по-малка стойност

ако не

b) с еквивалентна стойност по диагонала. [7]

Пример:

d_L		С	К	О	Б	Е	Л	Е	В
	0	1	2	3	4	5	6	7	8
С	1	0	1	2	3	4	5	6	7
К	2	1	0	1	2	3	4	5	6
Б	3	2	1	1	1	2	3	4	5
И	4	3	2	2	2	2	3	4	5
Л	5	4	3	3	3	3	2	3	4
И	6	5	4	4	4	4	3	3	4
Ф	7	6	5	5	5	5	4	4	4

Фигура 8. Пресмятане на разстоянието на Левенщайн между “скобелев” и “скблиф”

Забележка: Вертикално отместване следвано от хоризонтално такова и обратното добавя дупка в двете последователности и не е позволено.

Класа, който капсулира данните и методите по пресмятането на разстоянието на Левенщайн се нарича Levenstien:

```
class Levenstien
{
public:
   // Constructor and destructor
   .........

   int  LevenstienDistanceCalc(string T);
   bool InsertIntoTable(char Ch, int Column);
   int GetLevenstienDist(int Index);

   int GoodLevenstienDist;
```

```
   // intervals data
   int FirstColumns[10];
   int IndexFirstColumnArray;
   int PatternLength;                    // length of Pattern

private:
   int Minimum(int a, int b);

   string Pattern;                       // Pattern to search
   int (*LevenshtienDistanceTable)[M];   // Levenstien distance array
};
```

Метода int LevenstienDistanceCalc(string T) описва пресмятането на разстоянието на Левенщайн по описаната по-горе динамична схема. Този метод обаче не е използван директно в решението, тъй като в поставената задача нас ни интересува не колко е разстоянието на Левенщайн между две зададени думи, а дали то е по-малко или равно на някакво определено зададено число за близост k. Това означава, че за нашите пресмятания са важни $2k+1$ основни диагонала. Примерно, ако $k=2$, тогава таблицата на Левенщайн би имала вида на фигура 9. Останалата част от таблицата не е нужно да бъде пресмятана, защото тя няма отношение към това дали думите са на разстояние по-малко или равно на k. В този случай динамичната схема по пресмятането на Левенщайн разстоянието придобива следния вид:

$$d_L(\varepsilon,W)=|W|$$

$$d_L(V,\varepsilon)=|V|$$

$$d_L(aV,bW)=\begin{cases} d_L(V,W), & ако\ \ a=b \\ 1+\min(d_L(V,W),d_L(aV,W),d_L(V,bW)), & ако\ \ a\neq b\ u\ |aV|<k\ u\ |bW|<k \\ 1+\min(\textit{двете сметмати вече клетки}) \end{cases}$$

за $V,W\in\Sigma^*$ и $a,b\in\Sigma$.

Двумерният масив (*LevenshtienDistanceTable)[M] пази пресметнатите стойности за разстоянието на Левенщайн и е от тип int. Първият му индекс се мени по буквите (броя на буквите) на заявката, наречена шаблон, а вторият индекс се мени по буквите (броя на буквите) на прочетена дума от речника. Той обаче не се смята всеки път при достигане до крайно състояние, защото така би била загубена информация, която по-рано може да е била пресметната за друга дума, която присъства в речника, но е по-кратка от текущата. За да се избегне тази загуба на информация и повторното и пресмятане, таблицата не се пълни при прочитането на дума от речника, а колона по колона за всяка прочетена буква. В случай, че имаме връщане назад в речника просто се променя индексът на колоната, която трябва да се запълни, като се декрементира с едно. И така нататък.

d_L		t_1	t_2	t_3	..	t_k	..	t_m
	0	1	2	3	..	k	..	m
p_1	1				..		..	
p_2	2							
p_3	3							
..	..	..						
p_k	k							
..	..	..						
p_n	n							

Фигура 9. $2k+1=5, при\ к=2$ важни диагонала

Тази процедура по запълването на колона от таблицата с разстоянията на Левенщайн се извършва от член-метода bool InsertIntoTable(char Ch, int Column):

```
bool Levenstien::InsertIntoTable(char Ch, int Column)
{
   bool FlagValue = false;

   if(Column == FirstColumns[IndexFirstColumnArray])
   {
      InitializeInitial(Column);

      FlagValue = true;
   }
   else
   {
      // filling of the matrix LevenshtienDistance[][]
      for(int    i    =    (Column-FirstColumns[IndexFirstColumnArray])    -
      GoodLevenstienDist; i <= (Column-FirstColumns[IndexFirstColumnArray])
      + GoodLevenstienDist; i++)
      {
         if((i <= 0) || (i > PatternLength))
         {

         }
         else
         {
            char b = Pattern.at(i-1);

            // If there is a match between the last character of T and P
            if(Ch == b)
            {
               LevenshtienDistanceTable[i][Column]                          =
               LevenshtienDistanceTable[i-1][Column-1];
```

```
            }
            // If there is a match between the last character of T and P
            else
            {
               if(i   ==   (Column-FirstColumns[IndexFirstColumnArray])   -
               GoodLevenstienDist)
               {
                  LevenshtienDistanceTable[i][Column] = 1 + Minimum(
                  LevenshtienDistanceTable[i][Column-1],
                  LevenshtienDistanceTable[i-1][Column-1]);
               }
               else
               {
                  if(i  ==  (Column-FirstColumns[IndexFirstColumnArray])  +
                  GoodLevenstienDist)
                  {
                     LevenshtienDistanceTable[i][Column] = 1 + Minimum(
                     LevenshtienDistanceTable[i-1][Column],
                     LevenshtienDistanceTable[i-1][Column-1]);
                  }
                  else
                  {
                     LevenshtienDistanceTable[i][Column] = 1 + Minimum(
                     Minimum( LevenshtienDistanceTable[i-1][Column],
                     LevenshtienDistanceTable[i][Column-1]),
                     LevenshtienDistanceTable[i-1][Column-1]);
                  }
               }
            }

            if(LevenshtienDistanceTable[i][Column]  <  GoodLevenstienDist  +
            1)
            {
               FlagValue = true;
            }
         }
      }
   }

   return FlagValue;
}
```

Методът, който ни връща разстоянието на Левенщайн за дадена колона от Левенщайн таблицата (за ред не се говори тъй като е ясно, че става въпрос за последния ред от таблицата, който представя последната буква от заявения адрес), ако такова е било пресметнато, се нарича int Levenstien::GetLevenstienDist(int Index):

```
int Levenstien::GetLevenstienDist(int Index)
{
   ASSERT(Index > -1);
   ASSERT(Index < 256);

   ASSERT(FirstColumns[IndexFirstColumnArray] > -1);
   ASSERT(FirstColumns[IndexFirstColumnArray] < 256);

   ASSERT(IndexFirstColumnArray > -1);
   ASSERT(IndexFirstColumnArray < 256);
```

```
   if((PatternLength >= (Index-FirstColumns[IndexFirstColumnArray]) -
   GoodLevenstienDist) && (PatternLength <= (Index-
   FirstColumns[IndexFirstColumnArray]) + GoodLevenstienDist))
   {
      // result
      return LevenshtienDistanceTable[PatternLength][Index];
   }
   else
   {
      return -1;
   }
}
```

Тъй като в програмната система, която се реализира, говорим за адреси/части от адреси, трябва да се избере адекватна стойност за Левенщайн разстоянието като мярка за близост между два адреса (примерно стойност 2 или 3 би била абсолютно приемлива). Това обаче поражда следния проблем. В общия случай имената на улиците или кварталите не са еднодумни. Примерно улицата “Джеймс Баучър” се състои от две думи. В случай, че е избрано Левенщайн разстояние равно на три, а потребителската заявка е само думата “Баучър”, системата ще отговори че не съществуват приближения на този адрес с Левенщайн разстояние по-малко или равно на три, или ако даде някакъв отговор, то със сигурност той няма да включва улицата “Джеймс Баучър”, тъй като разстоянието на Левенщайн между “Джеймс Баучър” и “Баучър” е по-голямо от три (= 7).

Това означава, че отговорът на такава заявка трябва да бъде по-адекватен с вида на базата. В случай, че има питане за “Баучър” много по-интуитивно е да се изведе като резултат улица “Джеймс Баучър”. Това налага адресите в базата да се разглеждат на части, т.е. да се сравнява за разстояние на Левенщайн не целият адрес, а неговите съставни думи, като за разделител се ползва интервал ‘ ’. Тогава таблицата на Левенщайн разстоянието равно на 3 между заявката и съществуващия адрес ще има вида, показан на фигура 10.

По същият начин стои въпросът с многодумните потребителски заявки. Те се разделят на думи и се търси отговор на въпроса дали в базата съществува адрес, който съдържа заявените думи на даденото разстояние на Левенщайн и то в същата последователност. Първо се минава през първата дума. В случай, че е намерен частичен адрес, който я удовлетворява, по нататък се минава по търсене на следващата дума от зададената последователност. Ако и тя е намерена се минава нататък... В случай обаче, че обходената част от адрес не се доближава до заявената дума, частта от автомата по този клон не се обхожда за останалите думи от заявената последователност. Примерно, в базата има адрес “Княз Александър Дондуков”. При заявка от страна на потребителя за “Александър Дондуков” се извежда “Княз Александър Дондуков”. Ако заявената последователност обаче е “Петър Дондуков”, то в резултатите от претърсването

със сигурност адрес "Княз Александър Дондуков" няма да присъства.

d_L		Д	Ж	Е	Й	М	С		Б	А	У	Ч	Ъ	Р
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
Б	1	1	2	3	4			1	0	1	2	3		
А	2	2	2	3	4	5		2	1	0	1	2	3	
У	3	3	3	3	4	5	6	3	2	1	0	1	2	3
Ч	4	4	4	4	4	5	6	4	3	2	1	0	1	2
Е	5		5	4	5	5	6	5		3	2	1	1	2
Р	6			5	5	6	6	6			3	2	2	1

Фигура 10. Левенщайн разстояние $\kappa = 3$; заявката има вида "баучер", а адреса от базата "джеймс баучър".

Цялата тази алгоритмична схема е реализирана в метода void Requesting(int State, int Index, int PositionString). За всяка дума от заявката се построява един обект от класа Levenstien – LevenstienObj0, LevenstienObj1,..., като се съхранява тяхната последователност. Първоначално претърсването започва от началното състояние на автомата и с думата, която е шаблон на обекта LevenstienObj0. В случай, че в даден момент при достигане до ' ' е открито приближение на шаблона в LevenstienObj0, тогава обхождането на автомата в тази посока продължава, като вече търсения шаблон е този от LevenstienObj1... Ако не е намерено приближение за шаблона от LevenstienObj0, въобще не се стига до търсене на шаблона от levenstienObj1, което гарантира че при извеждане на близките резултати заявената последователност ще бъде спазена.

Подобен проблем се поражда и ако в заявката има съкращения. В случай че потребителската заявка има вида "Дж. Баучър" и разстоянието на Левенщайн е до 3, тогава ако съкращението не се разгледа като по-специален случай, в резултатните записи като приближение няма да присъства "Джеймс Баучър", защото разстоянието на Левенщайн между "Дж." И "Джеймс" е по-голямо от 3.

Едно възможно решение на този проблем е от заявката да се игнорират всички съкращения и да се осъществи търсене само по целите думи в нея. Това със сигурност ще изведе търсения адрес, но е възможно и да изведе някои излишни резултати. Във всеки случай това е по-приемливото като се има предвид същността на базата.

В базата присъстват много съкращения. Едни от тези съкращения са титлите на хората, на които е кръстен даден географски обект, примерно бул.“Ген. Михаил Д. Скобелев”. Тези записи най-вероятно са наложени от стандарт за изписване имената на географските. Но този стандарт ограничава много търсенето в базата, защото в случай че потребителя знае пълното име на личността, на който е кръстена улицата, примерно Михаил Дмитриевич Скобелев, и подаде такава заявка (или близка), резултатите от отговора няма да съдържат със сигурност “Ген. Михаил Д. Скобелев” ако разстоянието на Левенщайн е три примерно (думата “Дмитриевич” и думата “Д.” имат разстояние на Левенщайн 8). Същият е ефектът и ако потребителската заявка съдържа цялото наименование на титлата – “Генерал Скобелев”.

Вторият въпрос, въпросът с титлите на хората в имената на географските обекти, е разрешен като още при обработката на заявката всички титли от нея са игнорирани – ако присъства запитване за “ген. Скобелев” търсенето се прави само за “Скобелев” заради отстраняването на съкращенията, а ако има запитване за “генерал Скобелев”, търсенето се прави отново само за “Скобелев”, защото “генерал” е титла на човек, която в базата присъства като “Ген.” (тази обработка се прави в метода string OperateAbbreviationInRequest(string Request)). Всички тези съкращения на титли са записани в картата string AbbreviationMap[AbbreviationNumber][2]. След като съкращенията от заявката са игнорирани, за всяка дума от нея се проверява дали случайно не е някоя от обявените в AbbreviationMap карта, която в базата присъства във съкратен вид. Ако случайно е така, тя се разглежда като заявка със съкращение в титлата, т.е. съкращението се пропуска при търсенето.

Всички тези подготвителни действия са извършени в метода void RequestDictionary(string Request, int LevenstienDistance), който обработва заявения стринг и го привежда във вид, в който рекурсивната функция Requesting() може да осъществи претърсването:

```
void Dictionary::RequestDictionary(string Request, int LevenstienDistance)
{
   // if serialization is finished correctly
   if(FlagDictDeserialization == true)
   {
      string TranslatedWord = TranslateWord(LowerCase(Request));
      TranslatedWord = RemoveLastIntervals(TranslatedWord);

      // search for precise word
      if(LevenstienDistance == 0)
```

```
    {
        if(FindRequest(TranslatedWord) == TranslatedWord)
        {
            SearchedWords.push_back(TranslatedWord);
        }

        return;
    }

    string Word = "";

    for(int I = 0; I < TranslatedWord.length(); I++)
    {
        // find word from requested sequence
        if(TranslatedWord[I] == ' ')
        {
            // cteate new Levenstien object for every word from sequence
            Word = OperateAbbreviationInRequest(Word);
            if(!Word.empty())
            {
                LevenstienObj.push_back(new Levenstien(Word,
                LevenstienDistance));
                VecInt VecDisp;
                MarkersDisplay.push_back(VecDisp);
                VecInt VecFind;
                FindedDist.push_back(VecFind);
                Word = "";
            }
        }
        else
        {
            // remove requested abbreviations
            if(TranslatedWord[I] == '.')
            {
                Word = "";
            }
            else
            {
                Word.append(1, TranslatedWord[I]);
            }
        }
    }

    // add last word even it is empty
    Word = OperateAbbreviationInRequest(Word);
    LevenstienObj.push_back(new Levenstien(Word, LevenstienDistance));
    VecInt VecDisp;
    MarkersDisplay.push_back(VecDisp);
    VecInt VecFind;
    FindedDist.push_back(VecFind);

    // make request - begin recursion
    Requesting(InitialState, 1, 0);

    // free memory
    while(!LevenstienObj.empty())
    {
        LevenstienObj.pop_back();
        MarkersDisplay.pop_back();
        FindedDist.pop_back();
    }
}
```

```
   // if there is rpoblem with deserialization
   else
   {
      ProcessError("Couldn't deserialize dictionary");

      // mark execution error presentation
      FlagExecutionErrors = true;
   }
}
```

Методът string FindRequest(string Request) открива дали има точно съвпадение на запитването с някой от адресите в базата. Ако такова съвпадение има, понякога не е необходимо да се търси по-нататък, но това зависи от конкретните изисквания. В случая този метод се вика само когато изрично е казано разстоянието на Левенщайн да е 0. Тогава не се прави разделение на заявката на думи, нито пък се търси разстояние на Левенщайн, защото е излишно. Дава се единствено отговор на въпроса: "Принадлежи ли заявеният адрес на езика, разпознаван от детерминирания автомат, т.е. принадлежи ли заявеният адрес на базата от данни с адреси?".

Не на последно място, при първоначалната обработка на заявката стои въпросът за това дали всички букви в заявката са букви от азбуката на автомата/речника. По-съществен е моментът - дори и буквите да не са букви от азбуката, дали има букви от азбуката, на които те могат да отговарят. Такъв е случаят с латинските букви, някои от които могат да се интерпретират с кирилски букви. Такъв е и случаят с големите букви, които трябва да се разглеждат като малки. Тези проблеми се решават в член-функцията string TranslateWord(string T):

```
string Dictionary::TranslateWord(string T)
{
   string ResStr = "";
   int Index;

   // for all letters of T
   for(int I = 0; I < T.length(); I++)
   {
      if((int)T[I] < 0)
      {
         Index = (int)T[I] + 256;
      }
      else
      {
         Index = (int)T[I];
      }

      // assert if Ch is not ASCII code
      ASSERT(Index < 256);

      // if given symol is not present in the alphabet
      if(ASCIICodes[Index] == -1)
      {
         string ErrorMessage = "This symbol is not present in alphabet
         characters: ";
         ErrorMessage.append(1, T[I]);
```

```
            ProcessError(ErrorMessage);
         }
         // translate word
         else
         {
            ResStr.append(1, Characters[ASCIICodes[Index]]);
         }
      }

      return ResStr;
}
```

След като отговорите от заявката са готови, те трябва да бъдат изведени във файл – това е един от изходните параметри на системата. Името на този файл се задава като вход. Тъй като още при структурирането на резултатите, те са запазвани в списък, съхраняването им на външен носител е лесно. Тази процедура се извършва в метода void PrintGoodWords(char* ResFile):

```
void Dictionary::PrintGoodWords(char* ResFile)
{
   fstream FsOut = fstream(ResFile, ios::out | ios::trunc);

   if(FsOut.is_open())
   {
      // while there are good results
      // print them
      while(!SearchedWords.empty())
      {
         FsOut << UpperCase(SearchedWords.front()) << "\n";
         SearchedWords.pop_front();
      }
      FsOut.close();
   }
   else
   {
      string ErrorMessage = "Couldn't open result file ";
      ErrorMessage.append(ResFile);

      ProcessError(ErrorMessage);
   }
}
```

С този метод описанието на клас Dictionary е завършено.

5.1.3 Интерфейси

Класът, на базата на който е построен COM обекта и който дефинира неговите интерфейси, е CFdaObj.

Първата стъпка при осъществяване на връзка със сървъра е подготовката на речника. Това е процес, който трябва да се извърши еднократно и да подготви по-нататъшната работа на сървъра. Той е първото нещо, което трябва да бъде поискано от него. Интерфейсът, който предоставя тази услуга, е CreateMinAutomata(BSTR newInputFile, BSTR newInputTableName, BSTR newInputColumnName, BSTR newOutputFile, long*

newCheckResultCreate). Неговите пет входни параметъра имат следния смисъл:

- newInputFile е низ, който съдържа името на базата данни MDB, която трябва да бъде трансформирана в речник във вид на минимален краен детерминиран автомат;
- newInputTableName е низ, съдържащ таблицата, от която трябва да се извлекат адресите на географските обекти;
- newInputColumnName е името на колоната от горепосочената таблица, която съдържа списъка, от който трябва да се построи автомата (примерно колоната с улиците, или колоната с кварталите);
- newOutputFile е името на файла, където искаме речника да бъде сериализиран и съответно след това от него да зареждаме вече построения автомат;
- newCheckResultCreate е връщан резултат, съдържащ информация за статуса на изпълнението – дали се е извършило коректно или не.

Тук е мястото, където се създава обект от класа DictionaryCreation и където се извиква public метода му CreateMinAutomata(). След създаването на автомата, трябва да се извика метода за сериализация, за да бъде съхранен на твърд носител:

```
STDMETHODIMP CFdaObj::CreateMinAutomata(BSTR newInputFile, BSTR
newInputTableName, BSTR newInputColumnName, BSTR newOutputFile)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   //check and free, then copy
   SysFreeString(m_strInputFile);
   m_strInputFile= SysAllocStringByteLen((char*)newInputFile,
      SysStringByteLen(newInputFile));

   //check and free, then copy
   SysFreeString(m_strOutputFile);
   m_strOutputFile= SysAllocStringByteLen((char*)newOutputFile,
      SysStringByteLen(newOutputFile));

   //check and free, then copy
   SysFreeString(m_strTableName);
   m_strTableName = SysAllocStringByteLen((char*)newInputTableName,
      SysStringByteLen(newInputTableName));

   //check and free, then copy
   SysFreeString(m_strColumnName);
   m_strColumnName= SysAllocStringByteLen((char*)newInputColumnName,
      SysStringByteLen(newInputColumnName));

   _bstr_t bstrInFile, bstrOutFile;
   _bstr_t bstrInColumn, bstrInTable;

   *newCheckResultCreate = 1;

   bstrInFile = _bstr_t(m_strInputFile,true);
   bstrOutFile = _bstr_t(m_strOutputFile,true);
   bstrInTable = _bstr_t(m_strTableName,true);
   bstrInColumn = _bstr_t(m_strColumnName,true);
```

```
   ProcessError(".................................Called
   CreateMinAutomata.................................");
   DictionaryCreation *CreationDict = new DictionaryCreation((char
   *)bstrOutFile);
   CreationDict->CreateMinAutomata((char *)bstrInFile, (char *)bstrInTable,
   (char *)bstrInColumn);
   if(CreationDict->FlagExecutionErrors == false)
   {
      ProcessError("Dictionary creation completes successfully.");
      CreationDict->SerializeArray();
      if(CreationDict->FlagExecutionErrors == false)
      {
         ProcessError("Serialization completes successfully.");
         ProcessError("Server execution completes successfully.");
      }
      else
      {
         ProcessError("Serialization completes with errors.");
         ProcessError("Server execution completes with errors.");
         *newCheckResultCreate = 0;
      }
   }
   else
   {
      ProcessError("Dictionary creation completes with errors.");
      ProcessError("Server execution completes with errors.");
      *newCheckResultCreate = 0;
   }
   return S_OK;
}
```

Веднъж създаден и съхранен във файл, речникът може да бъде зареждан и използван наготово. Интерфейсът, който се извиква за да се зареди речника(автомата) и сървъра да бъде готов да отговаря на потребителските заявки, е LoadDictionary(BSTR newDictionaryFile). Единственото нещо, което той прави, е да създаде обект от класа за запитвания Dictionary и да извика неговия метод DeserializeArray() за десериализация на запазения във файл автомат. Този обект е данна на CFdaObj класа и е видим от всички негови интерфейси. Единственият входен параметър на LoadDictionary() е низът newDictionaryFile, който съдържа името на файла, в който речникът е бил сериализиран предварително.

```
STDMETHODIMP CFdaObj::LoadDictionary(BSTR newDictionaryFile)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   //check and free, then copy
   SysFreeString(m_strOutputFile);
   m_strOutputFile= SysAllocStringByteLen((char*)newDictionaryFile,
      SysStringByteLen(newDictionaryFile));

   _bstr_t bstrDictFile;

   bstrDictFile = _bstr_t(m_strOutputFile,true);

   *newCheckResultLoad = 1;
```

```
   ProcessError("................................Called
   LoadDictionary................................");
   Dict = new Dictionary((char *)bstrDictFile);
   Dict->DeserializeArray();

   if(Dict->FlagExecutionErrors == false)
   {
      ProcessError("Deserialization completes successfully.");
      ProcessError("Server execution completes successfully.");
   }
   else
   {
      ProcessError("Deserialization completes with errors.");
      ProcessError("Server execution completes with errors.");
      *newCheckResultLoad = 0;
   }
}
```

Със зареждането на речника, сървърът е готов да удовлетворява потребителски заявки. Интерфейсът, задействащ този механизъм, е Request(BSTR newRequestWord, long newLevenstienDistance, BSTR newResultsFile). Той именно извиква публичните методи на класа Dictionary за изпълнение на заявката и за връщане на резултата: RequestDictionary() и PrintGoodWords(). Този интерфейс трябва да бъде извикван при всяка отделна заявка към сървъра от страна на клиента. Входните му параметри имат следния смисъл:

- newRequestWord е низ с потребителска заявка;
- newLevenstienDistance е разстоянието на Левенщайн, на което е приемливо да се търсят близки до заявката думи;
- newResultsFile е дисковото мястото(файла), където да се запишат намерените резултати.

Дефиниция на Request()е много интуитивна:

```
STDMETHODIMP       CFdaObj::Request(BSTR       newRequestWord,       long
newLevenstienDistance, BSTR newResultsFile)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   //check and free, then copy
   SysFreeString(m_strResultFile);
   m_strResultFile= SysAllocStringByteLen((char*)newResultsFile,
      SysStringByteLen(newResultsFile));

   //check and free, then copy
   SysFreeString(m_strRequest);
   m_strRequest= SysAllocStringByteLen((char*)newRequestWord,
      SysStringByteLen(newRequestWord));

   _bstr_t bstrResultFile, bstrRequest;

   bstrResultFile = _bstr_t(m_strResultFile,true);
   bstrRequest = _bstr_t(m_strRequest,true);

   *newCheckResultRequest = 1;

   ProcessError("................................Called
   Request.................................");
   Dict->FlagExecutionErrors = false;
```

```
   Dict->RequestDictionary((char*)bstrRequest, newLevenstienDistance);
   if(Dict->FlagExecutionErrors == false)
   {
      ProcessError("Requesting completes successfully.");
      Dict->PrintGoodWords((char*)bstrResultFile);
      if(Dict->FlagExecutionErrors == false)
      {
         ProcessError("Results printing completes successfully.");
         ProcessError("Server execution completes successfully.");
      }
      else
      {
         ProcessError("Results printing completes with errors.");
         ProcessError("Server execution completes with errors.");
         *newCheckResultRequest = 0;
      }
   }
   else
   {
      ProcessError("Requesting completes with errors.");
      ProcessError("Server execution completes with errors.");
      *newCheckResultRequest = 0;
   }

   return S_OK;
}
```

При завършване работата на сървъра по обработване на потребителски заявки задължително трябва да се извика неговият интерфейс FreeDictionary(), който освобождава заетата памет при десераилизацията на речника:

```
STDMETHODIMP CFdaObj::FreeDictionary()
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   delete Dict;

   return S_OK;
}
```

След като клиентската програма знае за наличието и смисъла на тези интерфейси, тя може да ги извиква по подходящ за нея начин.

5.1.4 Управление на грешките

За всяка програмна система не трябва да се пропуска да се спомене и момента с изпълнението на програмата по некоректен начин, което може при неправилна обработка на грешките да доведе до срив на системата. Ето защо реализираната система предвижда и просто управление на грешките. То е концентрирано в модула ProcessingErrors, който съдържа само една функция void ProcessError(string ErrorDescription). Тя в случай на някакъв проблем, отваря файл по подразбиране, в който се извежда информация за дефекта – деня, часа и описание на грешката. Ето и кода на въпросната функция:

```
void ProcessError(string ErrorDescription)
{
```

```
   fstream LogFile("LogFile.txt", ios::out | ios::app);

   if(LogFile.is_open())
   {
      CTime CurrTime = CTime::GetCurrentTime();

      int Day = CurrTime.GetDay();
      int Month = CurrTime.GetMonth();
      int Year = CurrTime.GetYear();
      int Hour = CurrTime.GetHour();
      int Minute = CurrTime.GetMinute();
      int Second = CurrTime.GetSecond();

      LogFile << Day << "/" << Month << "/" << Year << " " << Hour << ":"
      << Minute << ":" << Second << " >> ";

      LogFile << ErrorDescription << "\n";
   }
}
```

Грешките по принцип могат да бъдат от всякакво естество, но конкретно в нашия случай те са породени или от некоректни входни данни, или от лош/непълен подбор на азбуката на автомата/речника. Ето защо информация относно тези случаи би било добре да се извежда на подходящо място, за да може да бъде анализирана и съответно да бъдат предприети някакви мерки. В конкретния случай, файлът с тази информация се нарича "LogFile.txt" и се помества на мястото, от където е стартиран клиентът, който достъпва нашето сървърно приложение.

5.2 Прост клиент

За достъп до реализирания сървър, трябва да има клиент от някакъв вид. Този клиент трябва да има следните характеристики:

- Да има графичен потребителски интерфейс, който е интуитивен и лесен за имплементиране;
- Библиотека, която е лесно разширяема, за да задоволи нуждите на COM сървъра.

За да може да се демонстрират възможностите на COM сървъра, което е създадено в отговор на поставените изисквания, е проектирано просто клиентско приложение. То е с интуитивен и елементарен интерфейс, следващ предоставените от сървъра възможности. Клиентът е базиран на MFC библиотеките на Microsoft Visual C++.

Основен критерий при бърза разработка на приложения е средата да е снабдена с механизъм за бързо изграждане на потребителски интерфейс на приложението. По този критерий беше избрано Microsoft Visual C++. Microsoft Visual C++ има добър функционален редактор на ресурсите на разработваното приложение. Това значително намалява времето за проектиране и настройка на потребителския интерфейс, особено като се има

предвид, че клиентското приложение има за цел да демонстрира функционалните възможности на системата за анализиране на пощенски адреси, а не да свидетелства добри дизайнерски умения.

Друг критерий, който предопредели разработката на клиентското приложение да бъде на Microsoft Visual C++, е съвместимият и стандартизиран достъп до ситемните ресурси на целевата операционна система. Производителността на създаденото приложение значително се увеличава, докато размерът на генерирания файл и използваната памет намаляват, когато абстракцията, която предлага средата, е по-близо до системния набор от функции. Microsoft използват цялостно абстракцията MFC (Microsoft Foundation Classes), която реално представлява класово представяне (представяне по класове) на Windows Application Program Interface (WinAPI 16/32). Чрез тази реализация се запазва производителността на приложението такава каквато би била ако бе написано на WinAPI, както и удобството от обектно-ориентираното представяне и управлението на обектите от страна на езика. Това подпомага намаляването на грешките и освобождаване на ресурсите при използването им.

Като се вземат предвид всичките тези точки, едно диалог-базирано приложение, използващо AppWizard и редактора на ресурси, е най-добрият избор за MFC клиент.

След като потребителският интерфейс е готов благодарение на Visual C++ Dialog Editor, за да бъде достъпено сървър приложението от клиента, е необходимо да се създаде и манипулира една инстанция на FdaObj сървърния обект в клиентското приложение. Това се осъществява или чрез добавянето на декларации на интерфейса в клиентското приложение или чрез импортването (#import директива). В нашия случай сме се спрели на втората възможност.

От клиентска гледна точка са важни четири основни стъпки за достъпването на COM обект:

- Инициализация на COM библиотеките;
- Получаване на CLSID на COM обекта;
- Създаване и използване на COM обекта;
- Деинициализация на COM библиотеките.

Както вече споменахме, клиентското приложение е диалог-базирано. Състои се от няколко класа, които управляват неговата функционалност. Като входна точка за всяко MFC базирано приложение служи класът CTheApp, който наследява CWinApp (модул SimpleClient). Първо трябва да се инициализират COM библиотеките. Тъй като клиентът използва MFC, за инициализация на COM библиотеките се използва AfxOleInit функцията. Най-доброто място за нейното извикване е функцията InitInstance от класа на приложението, а именно CTheApp.

InitInstance се извиква най-рано, при стартирането на приложението и преди да е направено инстанциране на COM обекта. Кодът, който инициализира COM библиотеките, е следния:

```
//Init OLE libraries and support
if (!AfxOleInit())
{
   AfxMessageBox("OLE initialization failed");
   return FALSE;
}
```

Основният диалогов прозорец на клиента съдържа две отделни части – менюто, заедно с "status" бара и "tool" бара от една страна, и диалоговия прозорец от друга. Класът CModelessMain (в модула Modeless) пази всякаква информация относно "status" бара и "tool" бара, зарежда диалоговите икони и създава диалоговия прозорец. Неговите основни задачи са: създаване на "status" бара и "tool" бара; показване на потребителските икони на диалоговия прозорец; обновяване на състоянието на "popup" менюто; обновяване на "status" бара в зависимост от състоянието на диалоговото меню; показване на "tooltip"-ове за "tool" бара и т.н.

Има специален модул, който предефинира MFC класове CDlgStatusBar и CDlgToolBar – dlgbars модула. Тези класове са направени с идеята да поддържат обновяването на статуса на елементите от потребителския интерфейс.

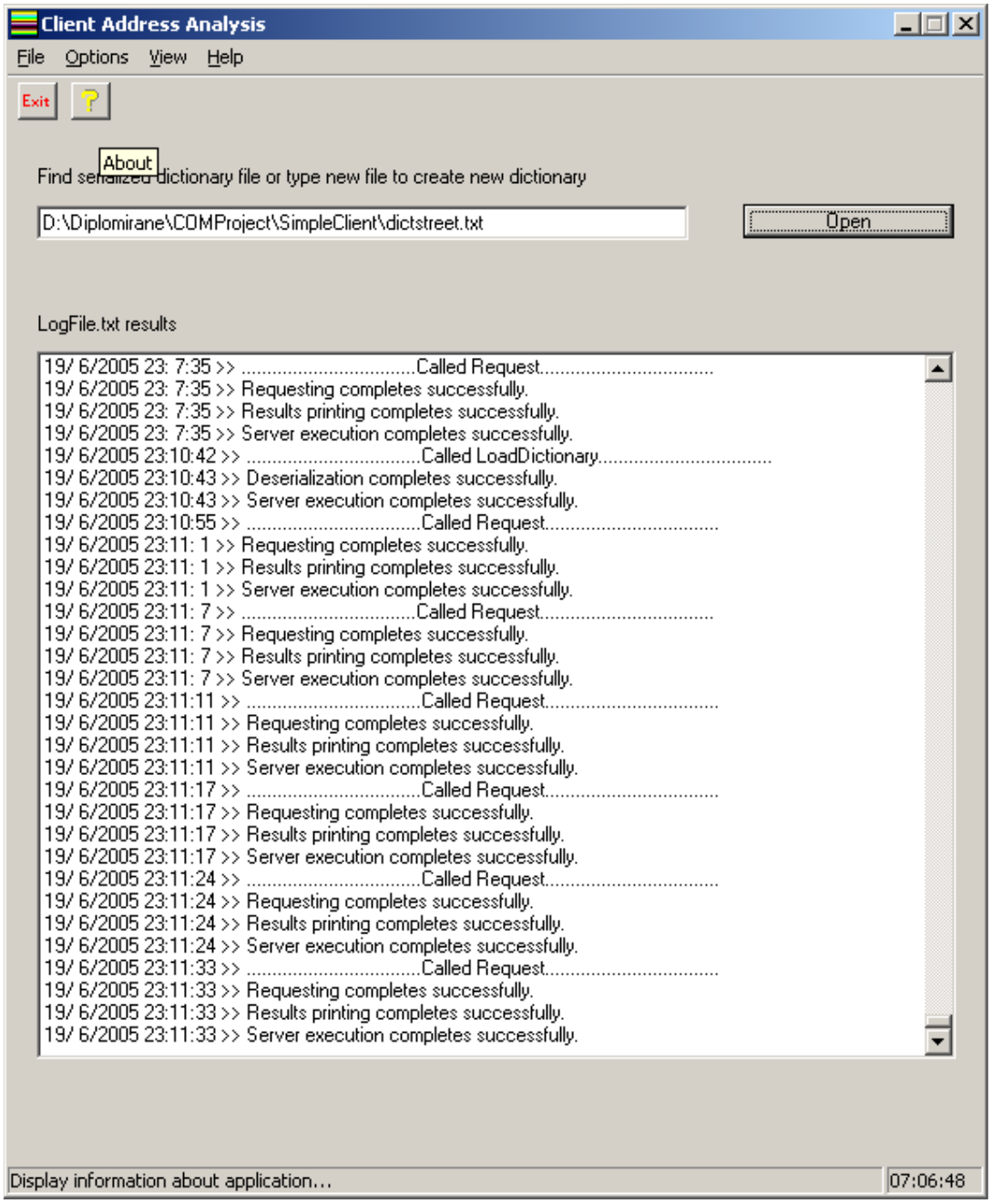
Класа на основният диалогов прозорец е CWndClientDlg в модула WndClient.

Основните меню елементи са няколко. Менюто "File" съдържа само един елемент – "Exit". С него се излиза от приложението. Следващото и най-съществено за нашето представяне е менюто "Options". То съдържа два поделемента: "Create Automata" и "Make Request". "Create Automata" отваря диалогов прозорец, който ни предоставя възможността да създадем речник от зададена база данни. "Make Request" създава диалогов прозорец, където потребителя може да задава своите запитвания за съществуването на даден адрес след като речник за всички адреси от този тип (в смисъл на тип е улица, квартал и т.н.) вече е построен. Менюто "View" ни дава възможност да изберем дали да се виждат или не "status" бара и "tool" бара, които по подразбиране при стартиране на приложението се виждат. Менюто "Help" дава информация за продукта.

Клиентският "Toolbar" се състои от два бутона – "Exit" и "About", които съответно излизат от приложението и дават информация за него.

Контролата за въвеждане по средата на клиентския диалог е мястото, където потребителя трябва да зададе файла с вече построеният речник в случай на предстоящо запитване за съществуването на адрес. Бутонът "Open" ще улесни

преглеждането за такъв файл. Ако такъв речник все още няма или ако има, но нас ни интересува създаването на нов, не е необходимо да се въвежда име на файл. В случай на създаване на речник и въведен файл, той може да бъде използван за мястото, където да бъде сериализиран речника.

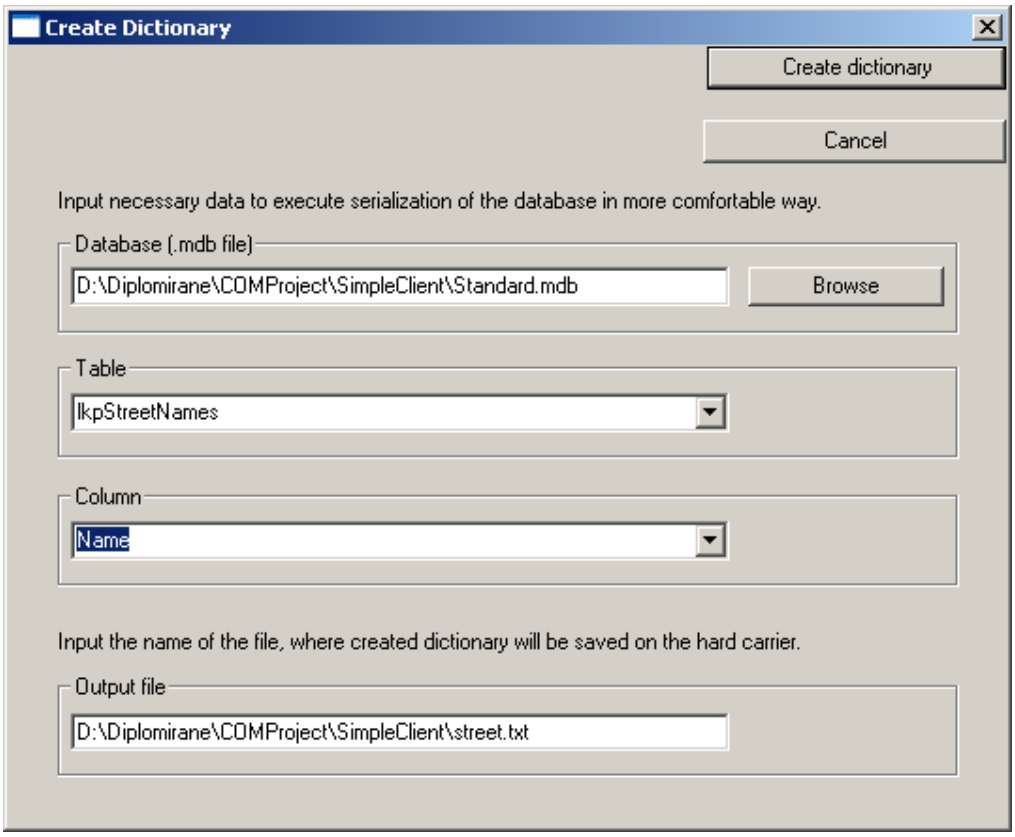


Фигура 11. Основен диалогов прозорец на клиентското приложение.

ListBox-а в долната част на диалоговия прозорец показва информацията, прочетена от "LogFile.txt" на сървър приложението, което, както беше уточнено по-горе, съдържа

информация за изпълнението на програмата на сървъра и за статуса на грешките, ако има такива. Той се обновява при затварянето на връзката със сървъра, а това се случая когато диалозите за създаване на автомат от база данни или за запитване, бъдат затворени.

Диалоговият прозорец за създаване на речник е управляван от класа CCreateDictionaryDlg в модула CreateDictionary.



Фигура 12. Създаване на речник.

Този диалог включва инстанцирането на COM обекта и връщането на уникален идентификатор за него. Общо разглеждани като CLSID (128 битов int), тези идентификатори специфицират COM обекта, който ще бъде инстантиран. Операционната система използва този номер да намери (разположи) компонентните файлове, използвайки системните регистри.

Един лесен начин да върнем това ID е чрез преминаване на програмното ID към CLSIDFromProgID функция. Тази Win32 функция взима програмното ID и търси в регистрите за асоциирано CLSID. Ако такова е намерено, то се връща. Именно този идентификатор е използван по-късно за създаването на инстанция на COM обекта. Тъй като клиентското приложение е диалог-базирано, най-доброто място за това е в OnInitDialog функцията. Тази функция се извиква преди диалога да стане видим, позволявайки

на приложението да се покаже бързо и да съобщи за неочаквани грешки. CLSID се връща със следния код:

```
hr= CLSIDFromProgID(OLESTR("AnalysisServer.FdaObj"), &clsID);
if(FAILED(hr))
{
   AfxMessageBox("Retrieval of ProgID failed");
   return FALSE;
}
m_pFdaObj.CreateInstance(clsID);
```

В този код програмното ID (в нашия случай AnalysisServer.FdaObj) е минало към извикване на функцията CLSIDFromProgID и върнатата стойност е тествана за възможни грешки. Веднъж получено CLSID, COM обекта може да бъде инстанциран.

"Edit" контролата в диалоговия прозорец позволява с помощта на бутона "Browse" да бъде избрана база от данни MDB. Благодарение на модула database тази база от данни се отваря и е възможно да бъдат прочетени таблиците й и колоните им. Те се зареждат в падащите менюта и позволяват избора да става по-лесно. В последното, най-долно "edit" поле се въвежда името на файла, в който искаме автомата речник да бъде запазен. По подразбиране това е файлът, който е бил прочетен от главния прозорец, ако такъв е бил избран.

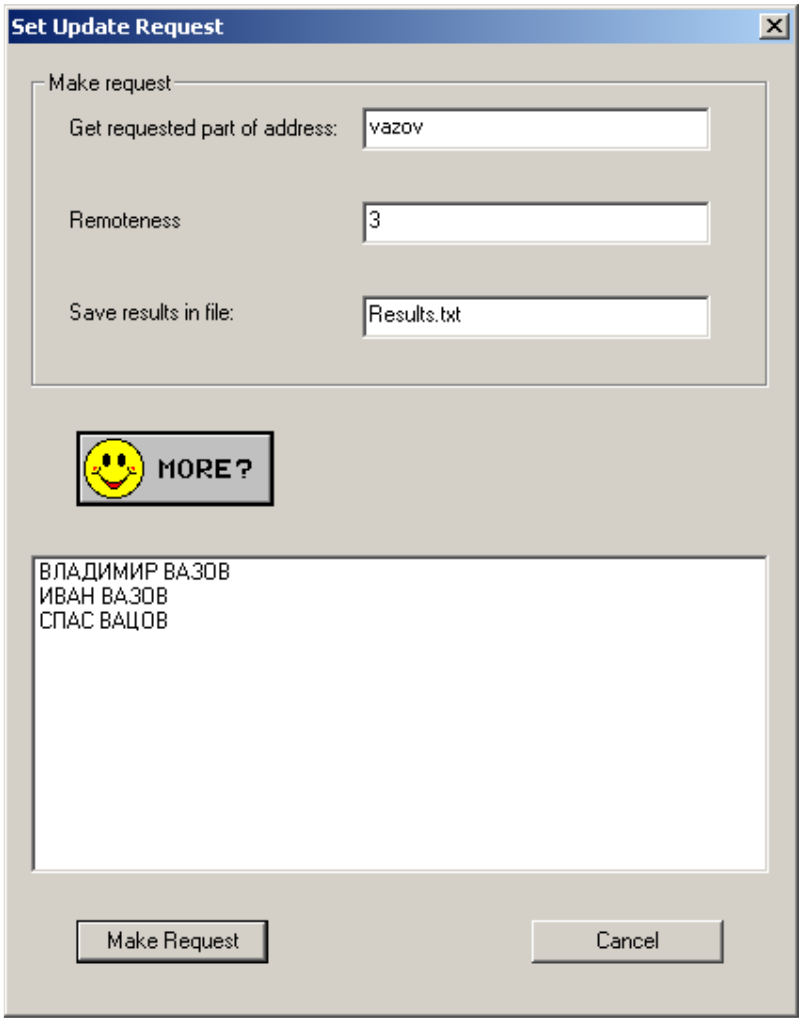
Бутонът "Create dictionary" създава речник като взима въведените данни за входни параметри. Основният момент в тази функция е обръщението й към интерфейса CreateMinAutomata с входните параметри от клиента. Първоначално се създава член-данна от тип IfdaObjPtr – типа на интерфейса, а след това чрез нея се вика интерфейсната функция на COM обекта. Изпълнението на създаването на речник завършва с отчитане на времето на изпълнение и статус на резултата от изпълнението.

Третият диалогов прозорец е свързан с възможността за извършване на запитвания за адреси. Класът, който реализира този диалог, е RequestDlg. При него също се създава член-данна от тип IfdaObjPtr, чрез която да се извикват интерфейсите на сървъра. Преди да могат да се извършват запитвания, речникът трябва да бъде зареден в паметта на компютъра. Това става с извикването на интерфейсния метод LoadDictionary, което е направено още при инициализиране на диалоговия прозорец в OnInitDialog. Това води до леко забавяне на появяването на диалога, тъй като в зависимост от размера на базата е необходимо известно време за десериализацията на речника. Преди това обаче е извикана CLSIDFromProgID функцията, за да върне CLSID на сървъра.

Първата "edit" контрола позволява да се напише запитване за дума, втората – да се зададе желаната мярка за близост (разстоянието на Левенщайн между думите) (по подразбиране това число е 3), а третата съдържа името на файла, където да бъдат

изведени резултатите. За удобство "list" контролата най-отдолу чете резултатите от файла с резултати и ги извежда. Но тъй като резултатите са в сортиран вид по разстоянието на Левенщайн, за удобство той ги извежда по следният начин: чете всички най-близки думи, които се разполагат в началото на файла и са с едно и също разстояние (т.е. първо извежда най-вероятните кандидати); ако все пак тези отговори не ни удоволетворяват може да поискаме още чрез бутона "More results". Така се извеждат думите с разстояние на Левенщайн с едно по-голямо(ако такива има) от предишната извадка. И така нататък.

Бутонът "Make Request" извършва запитването. Във функцията OnButtonMakeRequest, която се вика при натискането му, се извършва обръщение към интерфейса Request() на сървъра, и то с въведените входни параметри. Изход от това изпълнение е оценка за бързината на изпълнение и статуса на резултата – дали програмата е завършила коректно или не.



Фигура 13. Запитване за адрес.

Задължително, след като е зареден автоматът в паметта, трябва в края, когато приложението приключи работата си, да извика интерфейса FreeDictionary, който да освободи заделената памет.

6 Тестване и внедряване

6.1 Тестване коректността на реализацията

За да се убедим в коректността на реализацията на алгоритъма за построяването на речника от базата бяха направени някои характерни тестове. Най-добрият и сигурен тест си остава този с прочитането на всички думи, които извежда речника и дали те съвпадат с думите от базата, които са подадени първоначално. Това, че двете бази съвпадат, показва коректността на реализацията на алгоритъма на Михов. Обхождането на автомата беше направено с помощните член-функции void Dictionary::Traversing(int State) и void Dictionary::TraversingDictionary(). Първата функция е рекурсивна и прави пълно обхождане на автомата в дълбочина. При срещане на крайно състояние, прочетената дума е извеждана:

```
void Dictionary::Traversing(int State)
{
   if(Fin[State])
   {
      cout << "\n";
      PrintVisited();
   }

   for(int I = 0; I < CharMapL; I++)
   {
      if(Dict[State][I] != MissingTransition)
      {
         Visited.push_back(Characters[I]);

         Traversing(Dict[State][I]);

         Visited.pop_back();
      }
   }
}
```

Втората функция - TraversingDictionary() стартира рекурсията от началното състояние на автомата. След сравняване на изведения файл с началната база и съвпадането им, се убеждаваме в коректността на реализацията на алгоритъма за построяването на автомата, сериализацията му и десериализацията.

Остава въпросът с тестването на коректността на реализацията на динамичната схема за пресмятане на Левенщайн разстояние и коректното обхождане на базата, когато говорим за многодумни адреси и многодумни заявки. Вторият тест се прави на няколко стъпки. Първо се тества какво ще бъде изведено ако се зададе празна заявка и разстояние на Левенщайн, по-голямо от дължината на най-дългата дума в базата – в този случай системата извежда всички думи от базата, което само по себе си показва че базата се обхожда както трябва.

За това, дали реализацията на динамичната схема на Левенщайн смята коректно Левенщайн разстояние, са направени множество тестове за различни комбинации от думи. Може да се твърди, че след един набор от коректни тестове с различна природа, разстоянието на Левенщайн се смята правилно.

Остава обаче въпросът за това дали разстоянието на Левенщайн се смята коректно за многодумни заявки и многодумни адреси в базата, заради отместването на индексите и преизчисляването на стойностите. При анализиране на отговорите, които се извеждат, когато е подадена заявка, става ясно дали те са коректни. Следователно остава само въпросът дали това са всичките възможни отговори.

Един начин да се тества тази коректност е с много и различни тестове и бази, каквито бяха направени.

6.2 Performance тестове

Важни за използването на реализираната система са тестовете за производителност. Тези тестове са една от основните мерки, по които се мери качеството на един софтуерен продукт. Тестовете на разглежданата система са направени на компютър със следните хардуерни характеристики: CPU Intel Celeron 2.60GHz, Total Physical Memory 512MB, Operating System Microsoft Windows XP Professional Ver 5.01.2600 Service Pack 2. Измерването на времето за изпълнение се прави по елементарен начин с отчитане на времето точно преди стартирането на изпълнението на системата в клиентското приложение и отчитане на времето точно след приключване на изпълнението на системата отново в клиентското приложение, за да бъде обхванатa цялата функционалност – от извикването на някой от интерфейсните методи на COM компонента(сървъра) до връщането на резултата.

Най-важният критерий за производителност на разглежданата система е времето, а това от което този критерий зависи в нашия случай е размерът на базата с адресите. Таблиците по-долу показват какви резултати са получени при изпълнение на сървърната функционалност при различни по размер входни бази от данни за всяка различна реализирана функционалност.

Първоначално ще разгледаме създаването на речник от думите в базата от данни, което ще ни позволи бързото му обхождане по-късно. Необходимите ресурси за изпълнението на метода void DictionaryCreation::CreateMinAutomata(char* InFile, char* Table, char* Column) и сериализацията на информацията са дадени в таблица 2.

Ресурсите, които са необходими при осъществяването на запитвания са от най-голямо значение. При отговор на потребителска заявка са необходими ресурсите, описани в таблица 3.

Number records in .MDB file	CPU Usage	Mem Usage	Time execution
140	~0 %	~11000 K	0 sec.
2234	~55 %	~11000–12000 K	3 sec.
22890	~80-99 %	~13000-14000 K	45 sec.

Таблица 2. Изпълнение на интерфейса STDMETHODIMP CFdaObj::CreateMinAutomata(BSTR newInputFile, BSTR newInputTableName, BSTR newInputColumnName, BSTR newOutputFile)) – създава речник от база данни и съхранява създадения речник на твърд носител(сериализация)

Number records in .MDB file	CPU Usage	Mem Usage	Time execution
140	~0-2 %	~10000 K	0 ms.
2234	~0-2 %	~11000 K	15 ms.
22890	~10 %	~14000 K	78 ms.

Таблица 3. Изпълнение на интерфейса STDMETHODIMP CFdaObj::Request(BSTR newRequestWord, long newLevenstienDistance, BSTR newResultsFile) – запитване за съществуването на даден адрес

Може да се отбележи, че постигнатите резултати са сравнително добри.

6.3 Внедряване

За да може да бъде използван създаденият сървър е достатъчно DLL файла, който се генерира при build-а на COM обекта, да бъде регистриран/записан в регистрите. След което всяко клиентско приложение, знаещо интерфейсите му, може да се връзва към него и да му подава заявки.

Нашият COM обект е необходимо да бъде вмъкнат в системните регистри, така че той да може да бъде намерен от приложенията, които искат да го използват. За да бъде направено това е достатъчно просто да се направи регистрация на сървъра от командния ред чрез командата regsvr32 като се опише пълният път на 32 битовия сървър. На практика за инсталацията на сървъра е необходимо да се напише пътят на генерирания при build-а на сървъра DLL файл.

За да се улесни тази процедура е създаден инсталационният файл “AnalysisServerReg.bat”, стартирането на който от мястото, където се намира DLL-а, изпълнява тази регистрация.

7 Възможности за бъдещо развитие

По време на реализиране на програмната система и анализ на постигнатите резултати възникнаха няколко специфични възможности за оптимизации.

От една страна стои въпросът със съкращенията в заявката. По-добрият вариант може би би бил съкращенията да не се режат от заявките, а да се претърсва графът с речника само за адреси, които съдържат съкратените части в себе си и то така, както са написани, без да се допуска грешка в изписването на съкращението. В нашия случай обаче това едва ли би повишило много ефективността, защото единствено би намалило броя на възможните кандидати, като даже в случай че има правописна грешка в изписването на съкращението, би довело до пропускане на резултати, което от алгоритмична гледна точка е правилно, но от практическа няма да е съвсем адекватно на потребителските очаквания.

В този ред на мисли – когато говорим за съкращенията – възниква още една възможност. При претърсването на базата, когато в потребителската заявка присъства цяла титла, а в базата тази титла заради стандартите е съкратена, в текущата версия на реализацията не се търси по титлата, а се приема че тя е съкращение, защото в такъв вид присъства в базата, и се игнорира от заявката. По-приемливият вариант би бил да се търси по тази титла, като се разглежда като съкратена в заявката. Примерно когато заявката съдържа думата "генерал" в базата да се претърсват всички генерали (с титла в базата "Ген.") и само те. Това пък, от друга страна, би отрязало възможността, когато потребителят е направил грешка с коректната титла на личността, на която е кръстен обектът, но пък е изписал правилно името, да му даде отговор че такъв географски обект съществува, но с малко по-различно название на титлата. Има и възможност самата титла да се търси не конкретно като правилно изписана, а с по-специално разстояние на близост. Така възниква и идея за това една заявка да не се ограничава до едно разстояние на Левенщайн при търсенето, а за всяка част от нея да има специфично разстояние, примерно титлите да се търсят при едно разстояние на близост, а останалата част от заявката – при конкретно зададено.

Сега при претърсване на заявката за цяла титла, примерно "генерал", се гледа за цялостно коректно изписване на титлата, която се чете от списъка със съкращения в програмата. Може би по-уместно би било и тук да се търси не за думата "генерал" точно, а за някакви нейни модификации, на някакво разстояние на Левенщайн. Примерно, ако в заявката е изписано "генрал" да става ясно, че потребителя е имал предвид "генерал" и да търси за генерали.

В текущата реализация остава един нерешен проблем. Примерно, ако потребителят се интересува за съществуването на бул.”Ген. Михаил Дмитриевич Скобелев”, няма да му бъде изведен никакъв резултат в отговор на търсенето, защото в базата посоченият географски обект е изписан като “Ген. Михаил Д. Скобелев”.

Още една възможност за оптимизация би било списъка с съкращения на титлите, които присъстват в имената на географките обекти, но по стандарт са изписани съкратено в базата, да не са записани в програмата, а да се четат от някакъв външен файл, за да не се налага при промяна на базата да се пипа в програмата.

Един все още ненапълно разрешен случай е въпросът с връщаните резултати. В момента резултатите се връщат с големи букви. При сегашната версия на реализацията не е възможно те да се върнат във вида, в който съществуват в базата, защото информацията за това кои букви са изписани малки и кои големи е загубена още при построяването на автомата, тъй като всички думи се преобразуват в малки букви, за да е оптимален автоматът, и да се претърсва оптимално като не се взема предвид нито какви са буквите в заявката, нито в базата. Разбира се, изкуственото поставяне на главни букви в началото на всяка дума в наименованието на обекта, би решило проблема, но е точно толкова коректно, колкото и думите да се изписват изцяло с големи букви. И тъй като все пак има географски обекти, които не са имена на реално съществували исторически личности, а някакво съчетание от думи и освен първата дума в наименованието, останалите са изписани с малки букви, то и изкуственото поставяне на главни букви няма да е много приемливо. Ето защо подходът с връщането на резултата с големи букви изглежда достатъчно приемлив.

И не на последно място стои възможността за оптимизация на управлението на грешките. Практическото използване на системата вероятно ще подскаже въпроси, за които не сме се досетили. Те обаче едва ли ще предизвикат необходимост от дълбоки корекции в избраното решение и ще са на нивото на непосредствени, елементарни промени.

Програмата е проектирана и разработена така, че има възможност за използване на произволна база данни – просто като се смени модулът за прочитане на базата. Но поради ограничителното условие на фирмата възложител на задачата, която е основа на дипломната работа, базата да бъде MDB, тази реализация предполага и се грижи базата да е именно такава.

8 Изводи и заключения

Цялостно разработване от проучване и модифициране на скоро създадени алгоритми, тяхното реализиране и представянето на програмен продукт, за област, в която вече са направени голям брой програми и в която съществуват голям брой често използвани решения, макар и да се знае, че те са доста далеч понякога от оптималното, е интересно предизвикателство. Това прави реализираната система интересна не само от страна на осъществяване и представяне, но и на степен на постигнат успех на поставените цели.

Тъй като говорим за обширната задача за приближено съвпадение на шаблони (approximate pattern matching), която има широк кръг на приложение от една страна, но от друга страна за всяка конкретна задача е ограничена от входните условия, не съществува такова решение, което да покрие абсолютно оптимално всички възможности.

С предложената реализация бе представена методология за създаването на програмен продукт и съпътстващите дейности при неговото анализиране, проектиране и имплементиране. Теоретичната обоснованост на методологията е подкрепена от последните разработки в тази дейност. Практическата проверка за валидност е извършена по време на създаването на системата, с което се систематизират етапите на разработка на решението.

9 Използвана литература

[1] Stoyan Mihov, Minimal Acyclic Automata: Constructions, Algorithms, Applications, Ph.D. dissertation, Bulgarian Academy of Sciences.

[2] Bjarne Stroustrup, The C++ programming language – Special edition, 2000

[3] http://my.execpc.com/~gopalan/com/com_ravings.html

[4] Красимир Манев, Увод в дискретната математика, Второ издание, Нов Български Университет, София, 1998

[5] Klaus U. Schulz, Stoyan Mihov, Fast string correction with Levenshtein automata, International Journal on Document Analysis and Recognition, Springer-Verlag 2002

[6] Michael T. Goodrich, Roberto Tamassia, Data Structures and Algorithms in JAVA

[7] http://homepages.uc.edu/~baxtercs/Lecture1Pairwise.html

10 Приложения

1. CD, съдържащо кода на програмната система, документацията по темата, UML модел на системата, резюме на български и английски език.