BASIC IPHONE PROGRAMMING

Case: Dictionary Application

Mikko Kaijalainen

Information Technology Bachelor's Thesis

May 2010

SAVONIA-AMMATTIKORKEAKOULU

Koulutusohjelma

Informaatioteknologia (eng.)

Tekijä

Mikko Kaijalainen

Työn Nimi

Basic iPhone Coding, Case: Dictionary Application

Työn laji	Päiväys	Sivumäärä
Opinnäytetyö	18.05.2010	34

Työn valvoja	Yrityksen yhdyshenkilö
Arto Toppinen	Johtaja Max Bremer

Yritys

Wired-Bit Oy

Tiivistelmä

Tämän insinöörityön tavoitteena oli suunnitella, toteuttaa ja julkaista sanakirjasovellus Apple iPhone -älypuhelimelle.

Opinnäytetyön toimeksiantaja on Tamperelainen online-markkinointitoimisto, joka on pääasiassa erikoistunut verkkoviestintään, hakukoneoptimointiin sekä hakukonemainontaan.

Vaikka sisältö on yritetty selittää mahdollisimman kattavasti, opinnäytetyön lukijalla olisi hyvä olla aikaisempaa tietoa ohjelmoinnista. Työ sisältää perustiedot tarvittavasta välineistöstä sekä ohjelmistoista, joita tarvitaan sovelluksien tuottamiseen iPhonelle.

Ohjelmisto toteutettiin pääasiassa käyttäen Objective-C -ohjelmointikieltä sekä SQlite-tietokantaa. Työssä toteutettu ohjelmisto on julkaistu Apple Inc. omistamassa mobiiliohjelmistojen myyntiin tarkoitetussa App Store -verkkokaupassa.

Avainsanat

Objective-C, Mobiiliohjelmointi, iPhone

Luottamuksellisuus

Julkinen

SAVONIA UNIVERSITY OF APPLIED SCIENCES

Degree Programme

Information Technology

Author

Mikko Kaijalainen

Title of study

Basic iPhone Coding, Case: Dictionary Application

Type of project	Date	Pages
Thesis	18.05.2010	34

Academic supervisor	Company supervisor
Arto Toppinen	CEO Max Bremer

Company

Wired-Bit Ltd.

Abstract

The goal of the thesis was to design, implement and publish a dictionary application for Apple iPhone -smartphone.

The client for this thesis is a company named Wired-Bit ltd, located in Tampere, Finland, which specializes in network communication, search engine optimization and search engine advertising.

Even though the contents of this thesis have been tried to explain as simply as possible, it will be helpful if the reader has some knowledge of programming. Thesis includes basic information about hardware and software needed to produce applications for iPhone.

The application was mainly done by using Objective-C -programming language and SQlite database. The finished application has been published in App Store, an online shop for mobile software, owned by Apple Inc.

Keywords

Objective-C, Mobile Programming, iPhone

Confidentiality

Public

TABLE OF CONTENTS

EXPLANATION FOR ABBREVIATIONS USED

EXPLANATION FOR ABBREVIATIONS USED

API Application Programming Interface, a way for application to use system resources and libraries.

CPU Central Processing Unit, the 'brain' of all computers.

GUI Graphical User Interface, a user interface based on graphics like buttons, sliders, menus, etc.

IDE Integrated Development Environment, brings all the useful tools for coding into one place for the easy use by developer.

MVC Model-View-Controller, a design model which takes care of the application's data processing and display.

OS Operating System, a program which manages computer's resources (e.g Windows, Mac OS X, etc.).

SDK Software Development Kit, a pack of tools for developing new applications.

1. INTRODUCTION

1.1 Client

The client for my thesis is a company called Wired-Bit ltd, an online marketing office, which has been founded in 2007 and is operated from Tampere, Finland. The company specializes in network communication, search engine optimization and search engine advertising market.

1.2 Goals

The main goal of my thesis was to design, implement and publish a light and easy-to-use application for Apple iPhone -smartphone. The coding is done with Objective-C programming language.

Another goal for my thesis was to familiarize myself with mobile programming. The application was intended to be easy to convert into other similar applications with only slight modifications.

1.3 Useful literature

There is a huge amount of documentation available in internet to help with my thesis. For me the most important documentation was found from Apple iPhone developer page and from an e-book called “How to become an Xcoder”, which teaches basic coding with Objective-C language for Apple computers. This information is great for iPhone programming as well.

1.4 My background information

While starting to work with my thesis I did not have almost any idea about mobile programming. I think the topic is interesting and useful in modern programming so as my client suggested this topic for me, I wanted to give it a shot. In my studies I have had two useful courses considering my thesis: C-Programming and Java Programming, both of which have given me some basic information about object oriented programming.

2. PROGRAMMING FOR IPHONE OPERATING SYSTEM

2.1 Basic Information

iPhone OS is an operating system developed by Apple Inc. It was published in June 2007 and it is designed for use with touch screen. Beside iPhone, it is used also in iPod Touch music player and in iPad tablet computer. To make it easier to read, further on iPod and iPad are not mentioned in this thesis, but they are also referred to when mentioning iPhone.

Development of iPhone applications can only be done with Apple Inc. computers, although it is not possible with the oldest computer models.

2.2 Programming Language

The programming for iPhone is done by using Objective-C programming language. It is basically an object based extension for C-language which has all the basic C architectures included but also the classes and objects can be handled in the way done in other object oriented programming languages. The syntax of the language is still different, so only by knowing C-language the programming can not be done.

The application programming interface (API) used is Cocoa, which is recommended for new programs. It is based on Model-View-Controller (MVC, see fig 1.) design model. In MVC the program's functionality is divided to three areas, to which the application's classes are focused. As the model name suggests, these areas are called Model, View and Controller.

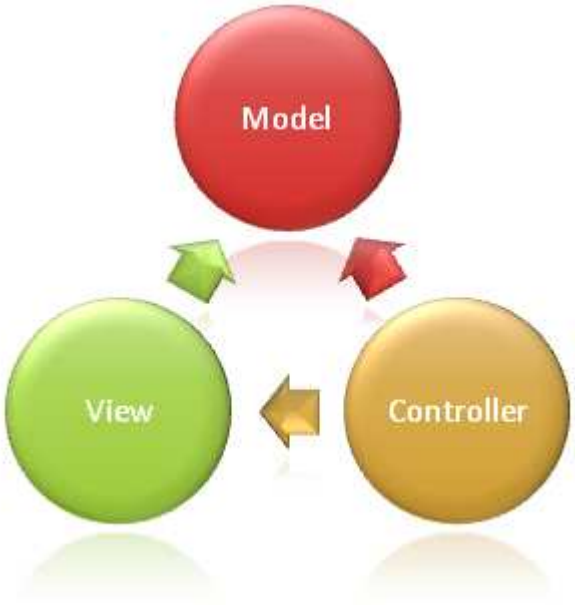

Figure 1: Model-View-Controller design model. /1/

The Model area classes take care of the data processing needed in the application, while the view area classes handle displaying of the data. The controller between them takes care of the actual programming logic. It processes the graphical user interface events performed by user, asks for the model area to make necessary changes in data and forwards the changed data for presentation in the view area. View area and model area are not discussing directly together, they always need the controller area between to do the 'talking'.

2.3 SDK (software development kit)

Apple Inc. published the software development kit for third party application developers in March 2008. The most important application development tools in SDK are: Xcode IDE, iPhone simulator, Instruments and Interface Builder. By knowing how to use these tools, it is possible to successfully develop iPhone applications.

2.3.1 Xcode IDE (Integrated Developer Environment)

Xcode IDE is a tool which is used to program an application by writing code. Application building and debugging are also done with this tool. I made my thesis application mostly by using this tool.

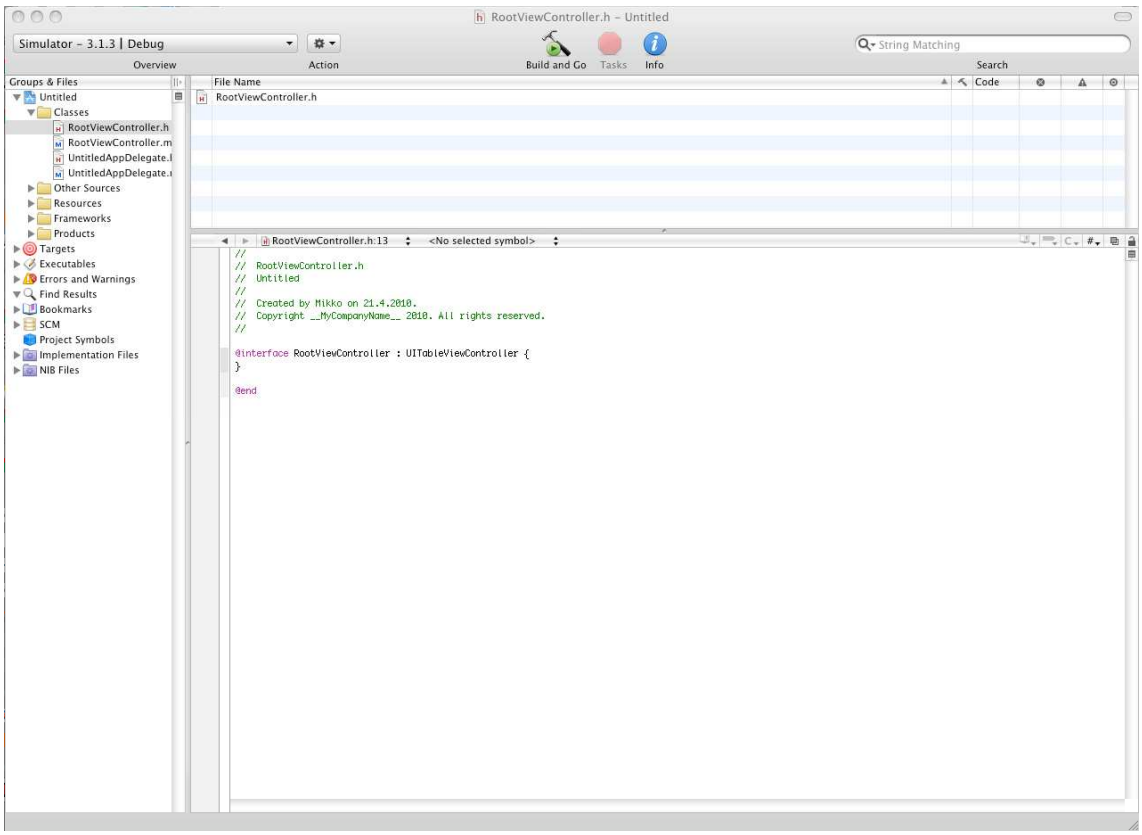

Figure 2: Xcode basic layout.

2.3.2 iPhone Simulator

IPhone simulator is used for application testing in a proper environment, although there is a minor problem while using it. The testing environment is much faster while testing on computer, compared to actual iPhone. Because of this it is important to test the application also in a real iPhone to make sure it works well.

The simulator is almost a perfect platform to test applications, including also touch screen gestures, tilting of the phone etc. Simulator also saves a lot of time cutting out the time spent in installing and testing the application in a real device.

Figure 3: iPhone simulator lookout.

2.3.3 Instruments

The tool named instruments tells important information about the application's performance. It gathers information about the usage of disc space, memory and central processing unit (CPU). The information gathered can also be seen in graphical form so it is easier to find possible memory leaks or reasons for application being slow.

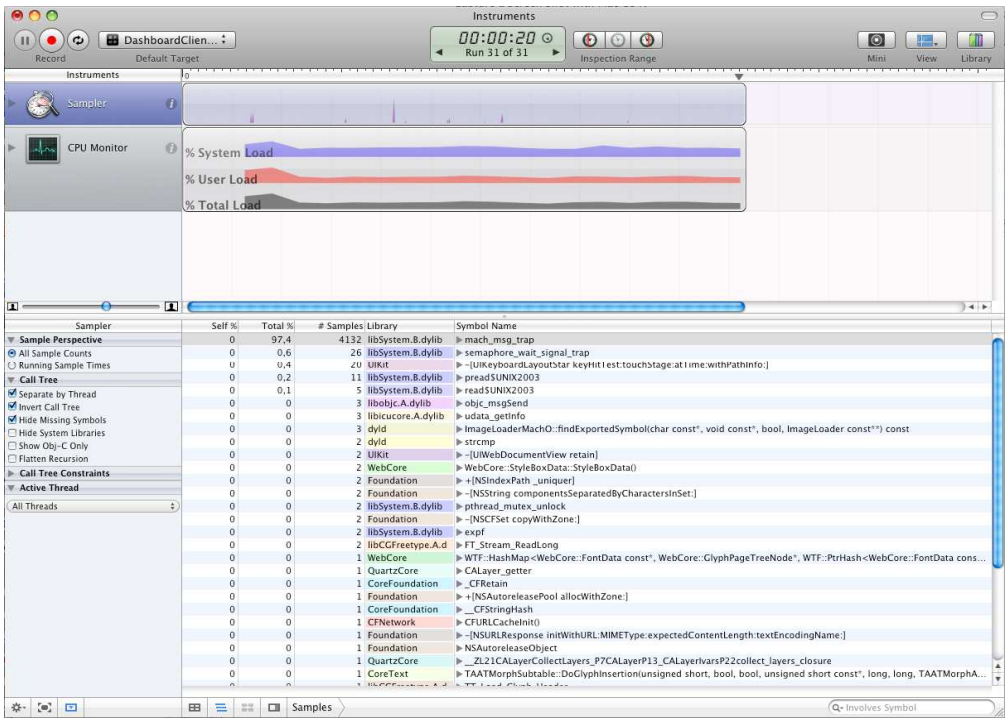

Figure 4: Instruments tool layout.

2.3.4 Interface Builder

As the name suggests, interface builder is used to construct a graphical user interface (GUI). By using interface builder can simple applications be done even without writing any code. In interface builder it is possible to graphically insert buttons, text fields etc. and connect events like button pressings to wanted actions. These all can also be done by writing objective-c code so it is not necessary to use interface builder at all.

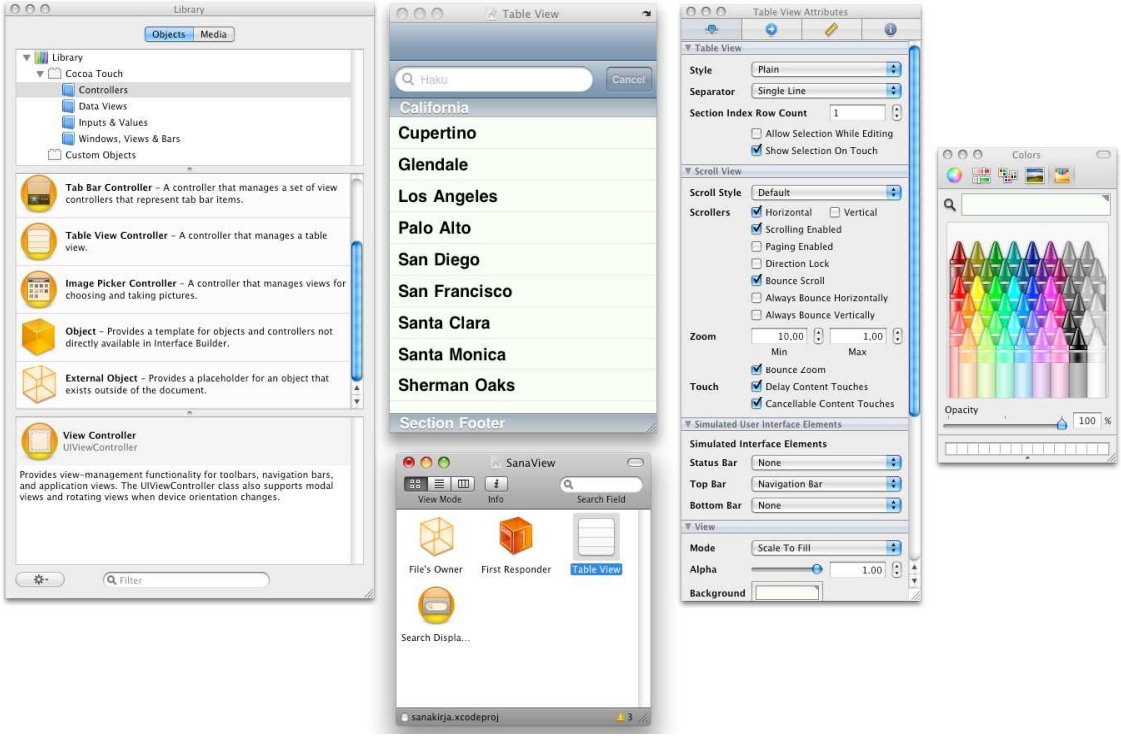

Figure 5: Interface builder lookout.

2.4 Database

The database in my application had to be lightweight so I chose SQLite. The best part of SQLite usage in mobile programming is that it does not need a separate server for database. Database can be included in the application. All the databases are written into single files, which normally is located in the device in use. Database files are normally really small in size and that is why SQLite is used in portable devices like mobile phones and mp3 players. SQLite is also free of charge.

The usage of SQLite is simple. After downloading a necessary command line tool, databases can be made in command prompt.

```
$ sqlite3 ex1
SQLite version 3.6.11
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table tbl1(one varchar(10), two smallint);
sqlite> insert into tbl1 values('hello!',10);
sqlite> insert into tbl1 values('goodbye', 20);
sqlite> select * from tbl1;
hello!|10
goodbye|20
sqlite>
```

Figure 6: An example of making a database and controlling it in SQLite. /2/

In the first row the actual database named 'ex1' is made with command 'sqlite3 ex1'. The next bold line includes the command 'create table tbl1' for making a database table. There are also two columns added to the table: 'one', whose type is varchar for a text string and column 'two' typed smallint for a integer value. On the next two lines the values are inserted into table columns, and after that those values are shown with command 'select'. The picture above is only a simple example of how to use SQLite, but in my application there is no need for more complicated commands.

In my thesis I had to insert a lot of information into my database so using the command line tool was not a good idea because of the workload included. Instead of writing commands myself, I used a free tool called 'SQLite Database Browser'. It contains a graphical user interface and makes it easier to add multiple lines of data into database. It still uses the same commands seen above, but user does not have to type them herself/himself. With this tool user can make new databases, tables and columns. Also browsing of database data graphically is possible as well as executing SQL commands.

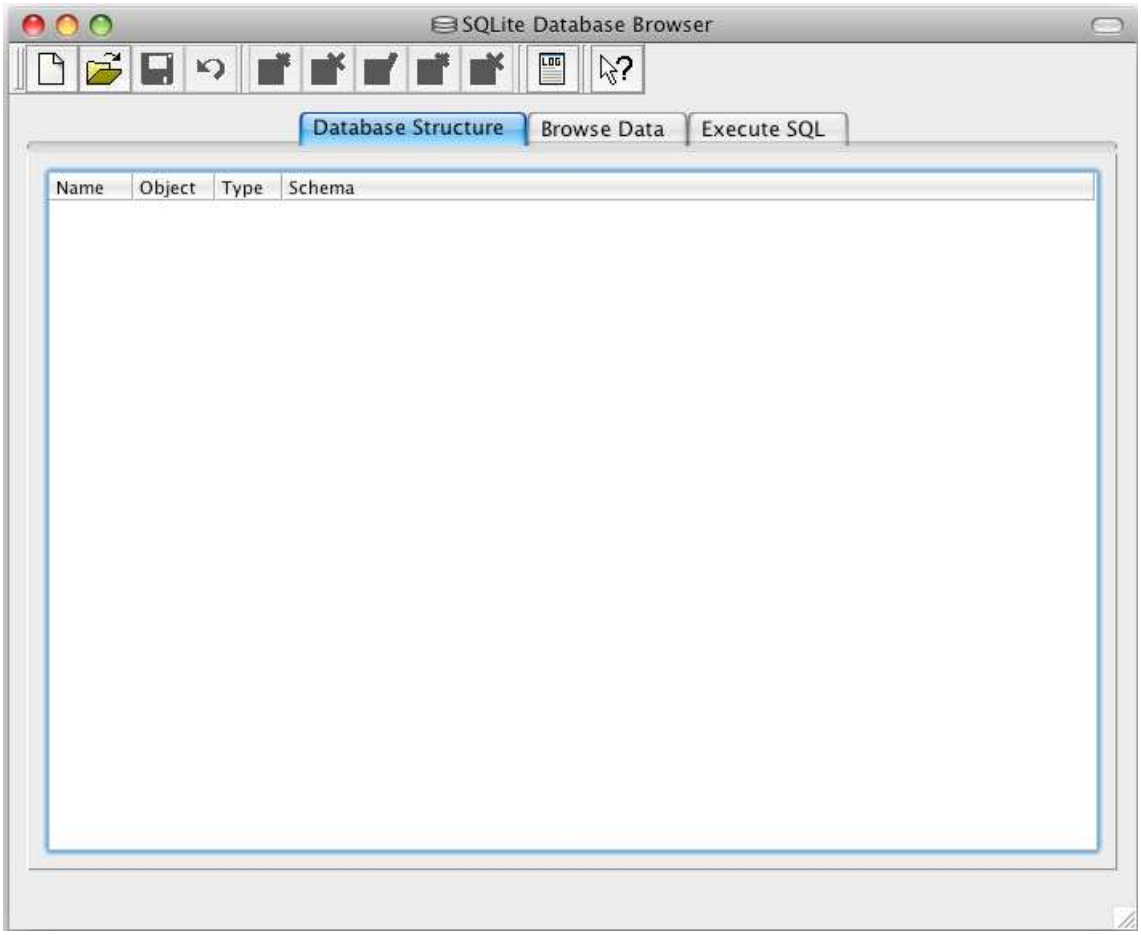

Figure 7: SQLite Database Browser lookout with no database open.

2.5 Getting Started

The development of a new application is started by opening Xcode and selecting 'new project'. In my thesis I used navigation-based application which is the easiest one when there are many views and no need for complicated graphics.

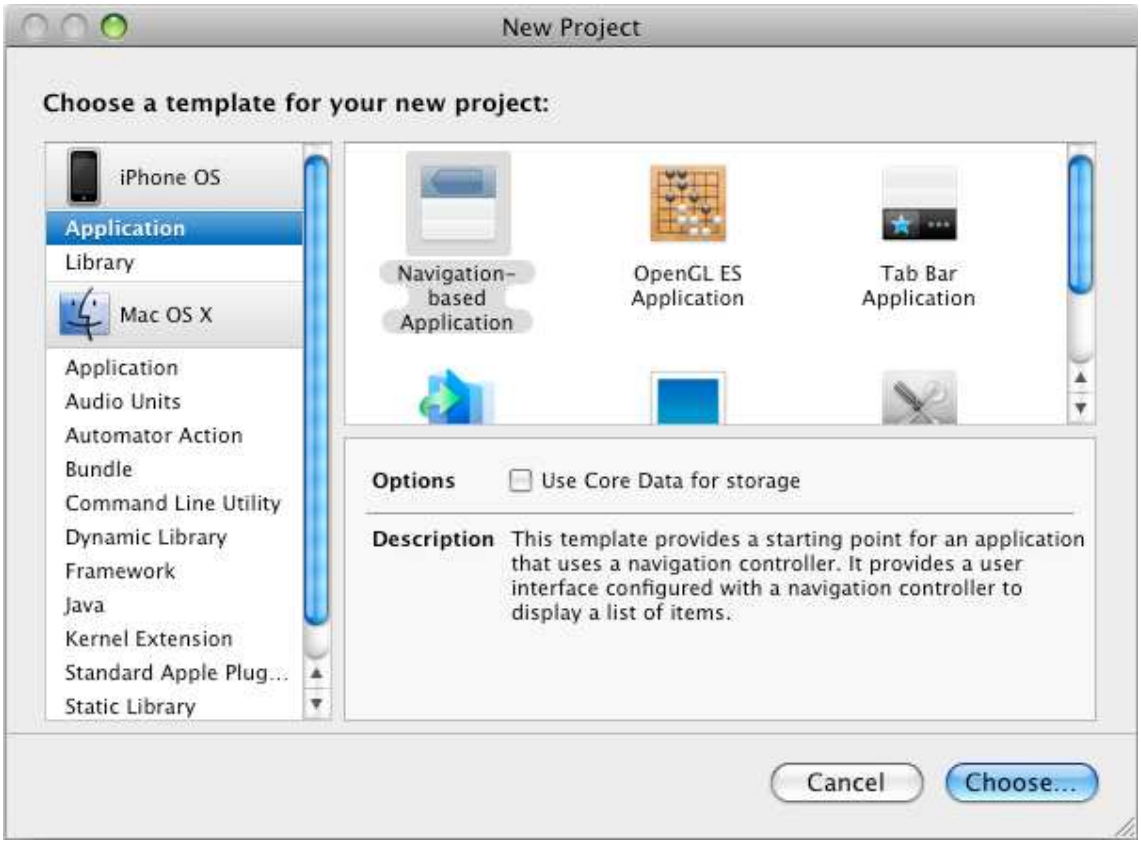

Figure 8: Starting view of Xcode IDE.

After giving a name to project in the next window, the main Xcode window (Fig. 8) will be visible.

2.6 Testing

When testing an application it is easy to use the tools described above, but it is also important to test the software in a real device. For this it is necessary to join Apple iPhone developer program, which costs 99 dollars for an independent user. After joining the program, user's device can be made into a test device which enables installing of new apps via Xcode IDE.

2.7 Distribution

Ready-made software can be sent to Apple App Store where it will be published for sale after verification. The developer can choose the price for his/her application but 30% of the profit is kept by Apple as a store upkeep fee.

3. APPLICATION PLANNING AND DESIGN

After having a conversation with my thesis client company's CEO, we had many

options for different applications. All of these applications were quite similar, including a database and a requirement for the application to be easy to use with touch screen. We ended up with a dictionary program between Finnish and Thai languages. One of this application's most important properties was to be easily modified into another application. That is why the views in software have to be simple and the database carefully chosen.

This application includes two different table views and a one normal view, between of whose can be moved by touching objects seen on the screen. Application also includes a SQLite database which contains all the dictionary words. This list contains Finnish words, their equivalent Thai words and their pronunciation, because Thai script is not readable for foreign people. The list of dictionary words in my application is taken from an internet forum and permission for use of the list has been asked from the author.

4. APPLICATION

4.1 Structure

Figure 9: Application's view structure.

My application constructs of the three different views seen in fig. 9. All of these views have to be coded separately, but they include many similarities. Apple Inc. has produced a few different base views for developers to make it easy to implement a well working application. These views are great for use with touch screen and it is recommended to use them.

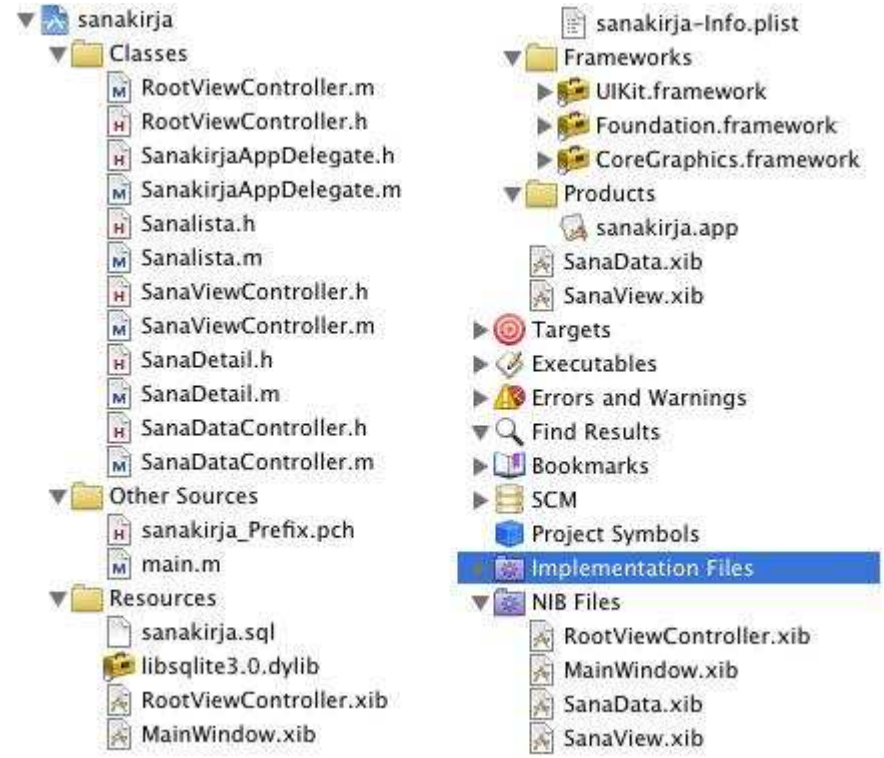

Figure 10: Files needed shown in Xcode IDE.

My iPhone application constructs of the files seen in fig 10. Files in the Classes folder are the main files where the actual code is written. The .h files are header files which contain our variable definitions. The .m files are invoked by main.m file and these files contain necessary code to make application work as desired.

Sanakirja_Prefix.pch file contains some data which will be included into frameworks. Normally developer does not have to edit this file. Main.m is the file where execution starts. There is no need to edit this file either.

In the Resources folder lays my database file sanakirja.sql, libsqlite3.0.dylib dynamic library file for SQlite database information and sanakirja-Info.plist file which contains some application initialization data.

Frameworks folder includes sets of library functions made by Apple to make it possible to code with certain functions. In simple applications there is no need to worry about these as they are automatically added.

The file sanakirja.app in products folder is the file that will be installed to user's iPhone.

NIB Files folder includes files from all of application's views. These .xib files contain visual information and can be edited via interface builder. Files ending with .xib will become .nib files when the application is builded and that's where the folder name comes from.

4.2 Views and graphical user interface

4.2.1 The application delegate

The AppDelegate is necessary in every application. It can be thought as a controller, which takes care of critical events while starting, running and ending application. In my project it is called SanakirjaAppDelegate. Below is my code for AppDelegate.

```
//sanakirjaAppDelegate.h
//#import command imports header file needed by the compiler to understand our code.
#import <UIKit/UIKit.h>
#import <sqlite3.h>

//Makes the compiler know what to expect (for example objects etc.).
@class SanaLista;

//@interface tells that this is a declaration
//of the class SanaKirjaAppDelegate.
@interface SanakirjaAppDelegate : NSObject <UIApplicationDelegate> {

    //IBOutlet makes it known that we can hook up things in Interface Builder.
    IBOutlet UIWindow *window;
    IBOutlet UINavigationController *navigationController;

    sqlite3 *database;
    NSMutableArray *SanaListaArray;
}
//with @property we declare how these properties behave.
@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) UINavigationController *navigationController;
@property (nonatomic, retain) NSMutableArray *SanaListaArray;

//void means these functions do not need to return anything now.
-(void)createDatabaseIfNeeded;
-(void)getInitialData;

//@interface always ends with @end.
@end
```

The green lines in my code parts are my own comments that are meant to help the reader to understand what is happening in code. There are lots of similar code parts like @property and @interface also in the coming code snippets so they will not be explained over and over again. Also some parts like #import commands are left out to make this easier to read.

The first code snippet is SanakirjaAppDelegate.h code which contains basic declarations and initializations for use throughout my application. The implementation file SanakirjaAppDelegate.m below does the correct procedures to include and start

using our database.

```
@implementation SanakirjaAppDelegate
@synthesize window;
@synthesize navigationController;
@synthesize SanaListaArray;
```

Implementation line will be automatically added when making a new program so it does not need to be worried about. @synthesize command is used to inform compiler about getter and setter methods for properties. These methods are used to control changes of a variable.

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    //When the application launch is finished, tells it to the delegate.
    //Checks if there is a database in iPhone already.
    [self createDatabaseIfNeeded];
    //Gets the initial data to the tableview.
    [self getInitialData];
    //Makes the main window to show up.
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

The method applicationDidFinishLaunching is a good place to make initial tasks and configurations before it has done anything else but started.

```
//Copies the database if it's not found on iPhone.
-(void)createDatabaseIfNeeded {
    BOOL success;
    //NSError object contains accurate error information.
    NSError *error;
    //NSFileManager lets you do operations in system files.
    NSFileManager *FileManager = [NSFileManager defaultManager];
    //Gets the list of right directories.
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
    //Get the first path in the array.
    NSString *documentsDirectory = [paths objectAtIndex:0];
    //Find the right path to our database file.
    NSString *databasePath = [documentsDirectory
    stringByAppendingPathComponent:@"sanakirja.sql"];
```

Finding the database file from user's iPhone. /3/

```
//Check if the file exists.
    success = [FileManager fileExistsAtPath:databasePath];
    //Quit if database is found.
    if(success) return;
    //If the database doesn't exist, we will copy it to the right directory in iPhone.
    NSString *dbPath = [[[NSBundle mainBundle] resourcePath]
    stringByAppendingPathComponent:@"sanakirja.sql"];
    success = [FileManager copyItemAtPath:dbPath toPath:databasePath error:&error];
    //If there is a problem, show error message.
```

```
    if(!success)
      NSAssert1(0, @"Copying database failed (%@).", [error localizedDescription]);
}
```

More initializations on database location. /3/

```
//Get the initial data from the table 'menu' in the database. This is not needed in
//my application, but it's there for the further use of the program.
- (void)getInitialData {
    //Allocating memory for array
    self.SanaListaArray = [[NSMutableArray alloc] init];
    //Find the database
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    NSString *dbPath = [documentsDirectory
    stringByAppendingPathComponent:@"sanakirja.sql"];
    //Open the database
    if(sqlite3_open([dbPath UTF8String], &database) == SQLITE_OK)
    {
      const char *sql = "select id, nimi from menu";
      sqlite3_stmt *selectStatement;

    //Sqlite3_prepare_v2 command makes sure that the database is connectable. If
    //returned value is SQLITE_OK, everything went fine, otherwise there will be an
    //error message.
      int returnValue = sqlite3_prepare_v2(database, sql, -1, &selectStatement, NULL);
      if(returnValue == SQLITE_OK)
      {
            //Get all the rows in database with this loop
            while(sqlite3_step(selectStatement) == SQLITE_ROW)
            {
                  //Get the first column value to be the primary key and store the
                  //next value to the SanaNimi string.
                  int key = sqlite3_column_int(selectStatement, 0);
                  NSString *SanaNimi = [NSString stringWithUTF8String:(char
                  *)sqlite3_column_text(selectStatement, 1)];

                  //create an object named SanaLista and fill it with values of the
                  //rows
                  SanaLista *aSanaLista = [[SanaLista alloc] initWithSanaData:key
                  SanaNimi:SanaNimi];
                  [SanaListaArray addObject:aSanaLista];
                  [aSanaLista release];
            }
      }
      //Release memory used for this loop.
      sqlite3_finalize(selectStatement);
    }
    //Close the connection to the database.
    else
      sqlite3_close(database);
}
```

This long code part is needed to get all the values from the database to be used in the application. Here every row of the database will be gone through and saved into the

object SanaLista. After this the database will be closed and is no more needed in our application. /3/

```
//Tell the delegate when the application is about to terminate.
- (void)applicationWillTerminate:(UIApplication *)application {

    //Close the database connection
    if(sqlite3_close(database) != SQLITE_OK)
    NSAssert1(0, @"Could not close the connection to database (%s).",
    sqlite3_errmsg(database));
}
```

If there is need to do some cleaning up before the application will be closed, this is the right code part. Cleaning up can include for example memory freeing.

```
//When the object is being removed from the memory, 'dealloc' is called.
- (void)dealloc {
    [navigationController release];
    [window release];

    //This line is important. If we don't use command 'super dealloc', the object isn't
    //removed and can cause problems with memory usage.
    [super dealloc];
}
//Ending the declaration of the class.
@end
```

These commands are executed when the application is exiting.

The code seen here was all the code needed in my AppDelegate. These were initialization tasks of my program to ensure good performance later on in my application.

4.2.2 Rootviewcontroller

As the name tells, RootViewController is the first view controller of the application, which contains the data shown to the user when he/she starts an application. In file RootViewController.h there are only a few lines of code needed to explain. Other lines are similar to code seen before.

```
UIButton *button;
```

There is only one button in my applications first view. The button is full screen size and not visible. If the user clicks the screen, next screen will load. If user does not do anything, the load will load automatically in three seconds. (see fig 11).

Figure 11: Application's main view.

```
@property (nonatomic, retain) IBOutlet UIButton *button;
//When button pressed, go to next view.
- (IBAction)goToSanaView;
```

This IBAction declares what will happen when user interface button is pressed. The action goToSanaView is declared in file RootViewController.m.

```
- (IBAction)goToSanaView {
    if(SvController.nibName == nil)
    SvController = [[SanaViewController alloc] initWithNibName:@"SanaView"
    bundle:nil];
```

Here the method for action goToSanaView is declared. When the button is pressed, proper view controller is loaded. In this case it will load the first table view (shown in fig. 13). The name of the view controller nib file (SanaView) must be declared.

```
    //The line below shares the SanakirjaAppDelegate to be used from all the other
    //classes. That way we can put global variables there.
    //It also holds important information about the initialization of the application
    SanakirjaAppDelegate *appDeleg = (SanakirjaAppDelegate *)[[UIApplication
    sharedApplication] delegate];

    //Gets the first value for a list needed in SanaViewController.
    SvController.aSanaLista = [appDeleg.SanaListaArray objectAtIndex:(1)];
    //Make SanaViewController visible.
    [self.navigationController pushViewController:SvController animated:YES];
```

Initializing a new view.

```
-(void)viewDidLoad {

        self.navigationController.navigationBarHidden = YES;
        self.title = @"Menu";
    [self performSelector:@selector(waitTime) withObject:nil afterDelay:1];
```

There is no need for navigation bar (see fig. 12) to be shown in the first window. The function waitTime is called after one second.

Figure 12: Navigation bar.

```
- (void)waitTime {
    if(SvController.nibName == nil)
    [self performSelector:@selector(goToSanaView) withObject:nil afterDelay:2]; }
```

If the user has not touched the screen, goToSanaView function has not been called. Then wait two seconds before calling that function to go to next view.

```
- (void)viewWillAppear:(BOOL)animated {
    self.navigationController.navigationBarHidden = YES;
    [super viewWillAppear:animated];
}
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
}
- (void)viewWillDisappear:(BOOL)animated {
}
- (void)viewDidDisappear:(BOOL)animated {
}
```

These methods can be called if developer wants the view to appear animatedly.

```
-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientati
on {
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; }
```

In this application there is no need for orientation support, because it might be inconvenient in use. That's why the orientation is forced to be portrait. In didReceiveMemoryWarning method we can release anything to free up memory.

4.2.3 First tableview

Figure 13: Tableview shown for user to browse through dictionary words.

After SanaViewController is requested from the previous view, the new view will be loaded. It is the first table view used in this application and it contains all the dictionary words. In the header file there are two arrays declared.

```
    NSMutableArray *ListaSanat;
    NSMutableArray *filteredListContent;
```

The first array called ListaSanat will include every word and the other array filteredListContent will be used to hold some words when user uses the search function.

```
//Static tells the compiler that *SanaValinta_statement is a global variable
static sqlite3_stmt *SanaValinta_statement = nil;
@implementation SanaViewController
@synthesize aSanaLista, ListaSanat, filteredListContent, searchWasActive;

//ViewDidLoad is called when the view controller has loaded the views into the memory.
- (void)viewDidLoad {
    //Make search function invisible
    [self.searchDisplayController setActive:NO animated:YES];
    self.searchDisplayController.searchBar.hidden = TRUE;

    //Make a search button to the top right corner of the view.
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc]
```

```
    initWithBarButtonSystemItem:UIBarButtonSystemItemSearch
                                            target:self
                                            action:@selector(searchBar:)];

    //Keep navigation bar visible.
    self.navigationController.navigationBarHidden = NO;

    //Give the title for the view.
    self.title = @"Sanat";

    //Enable scrolling.
    self.tableView.scrollEnabled = YES;
    [self.tableView reloadData];
```

```
    //Make the view to show the first cell.
    [self.tableView scrollToRowAtIndexPath:[NSIndexPath indexPathForRow:0 inSection:0]
    atScrollPosition:UITableViewScrollPositionTop animated:NO];
```

Even though there is a search bar in the application, there is no need for it to be visible all the time. Without this method it would be shown all the time, because table view makes the search bar go be the on the first row. This method skips that row. Search bar will be seen when user wants to see it by pressing search button.

```
    //Setting background color.
    [self.searchDisplayController.searchResultsTableView setBackgroundColor:[UIColor
colorWithRed:.600 green:.729 blue:.953 alpha:1]];
}
```

This is one way if we want to give a color to our table view. This can also be done in interface builder.

```
- (void)viewWillAppear:(BOOL)animated {
    self.navigationController.navigationBarHidden = NO;

    //Making superview animated.
    [super viewWillAppear:animated];
}
```

Now it is better to have navigation bar visible again.

```
//Actions when pressing the search button.
-(void)searchBar:(id)sender{
    [self.searchDisplayController setActive:YES animated:NO];
    self.searchDisplayController.searchBar.hidden = FALSE;
    [self.searchDisplayController.searchBar becomeFirstResponder];
}
```

Making search bar and keyboard appear when user presses search button. The becomeFirstResponder method saves some time by making also keyboard automatically appear. Without it, user would need to press two times for keyboard to appear.

```
- (void)viewDidUnload
{
```

```
    self.filteredListContent = nil;
}
```

Clear the filteredListContent array.

```
-(void)setASanaLista:(SanaLista *)ch {
    if(aSanaLista == nil)
    aSanaLista = [[NSMutableArray alloc] init];
    aSanaLista = ch;
    [self getList0fAllSanatForThisSanaLista:aSanaLista.SanaListaId];
}
```

Initialize an array for use with database

```
//Make a function which will get all the words from database.
- (void)getList0fAllSanatForThisSanaLista:(NSInteger)SanaListaId {
    self.ListaSanat = [[NSMutableArray alloc] init];
    self.filteredListContent = [[NSMutableArray alloc] init];
    sqlite3 *database;
```

Previously there was a same type of function to get words from database. Now the actual dictionary words will be fetched so there has to be a new function. This is quite similar to the previous one, so some of the code lines are left out. /4/

```
    //0pen database.
    if(sqlite3_open([dbPath UTF8String], &database) == SQLITE_0K)
    {
      const char *sql = "select suomi, thai, lausunta from sanalista";
      int returnValue;
      if(SanaValinta_statement == nil)
      returnValue = sqlite3_prepare_v2(database, sql, -1, &SanaValinta_statement,
      NULL);
      sqlite3_bind_int(SanaValinta_statement, 1, SanaListaId);
      //Loop through every row.
      while(sqlite3_step(SanaValinta_statement) == SQLITE_R0W)
```

The database table includes columns suomi, thai and lausunta and those values will be taken from the database here. /4/

```
            //Add the data to the object.
            SanaDetail *sd = [[SanaDetail alloc] init];
            //Get values from all three columns.
            sd.suomi = [[NSString stringWithUTF8String:(char
            *)sqlite3_column_text(SanaValinta_statement, 0)] copy];
            sd.thai = [[NSString stringWithUTF8String:(char
            *)sqlite3_column_text(SanaValinta_statement, 1)] copy];
            sd.lausunta = [[NSString stringWithUTF8String:(char
            *)sqlite3_column_text(SanaValinta_statement, 2)] copy];

            //Add the object to our array.
            [self.ListaSanat add0bject:sd];
            [sd release];
```

The values got from database will be entered to object sd. /4/

```
-(NSInteger)tableView:(UITableView *)tv numberOfRowsInSection:(NSInteger)section {
     //Defines how many rows we need in our table view to show all the information.
     if (tv == self.tableView)
     {
        return [self.ListaSanat count];
     }
     else
     {
       return [self.filteredListContent count];
     }
```

The declaration of table view cells starts here. If the table shown for user is the normal table view (see fig. 13), the amount of cells for words will come from ListaSanat array row count. If the table view is not that, in this case meaning search result table view (see fig. 14), the amount of cells will come from search list instead. /5/

Figure 14: Search result table view.

```
//Makes the cells to be reusable.
-(UITableViewCell *)tableView:(UITableView *)tv cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
     static NSString *cellNr = @"cell number";
     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cellNr];
     if (cell == nil)
     {
       cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
```

```
reuseIdentifier:cellNr] autorelease];
        cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
        self.tableView.separatorColor = [UIColor orangeColor];
        cell.textLabel.font = [UIFont fontWithName:@"GeezaPro-Bold" size:22];

    }
```

We want the separator line color to be orange so that it is visible with black background. Text font is also changed. /5/

```
    //Fills the cells with dictionary words.
    SanaDetail *sd = nil;
    if (tv == self.tableView)
    {
       sd = [self.ListaSanat objectAtIndex:indexPath.row];
       cell.textLabel.text = sd.suomi;
   }
    else
    {
        sd = [self.filteredListContent objectAtIndex:indexPath.row];
       cell.textLabel.text = sd.suomi;
   }
```

This looks similar to the earlier function and it works the same way. If the table shown for user is the normal table view, the list of words will be ListaSanat which includes all words. If the table view is not that, the list of words will be searched list instead. /5/

```
//Makes the search function work by reloading the table view after every entered letter
in search.
- (BOOL)searchDisplayController:(UISearchDisplayController *)SdController
shouldReloadTableForSearchString:searchText {
    [self filterContentForSearchText:searchText];
    return YES;
}
```

```
- (void)filterContentForSearchText:(NSString*)searchText {
    [self.filteredListContent removeAllObjects];

    for (SanaDetail *sd in ListaSanat) {
    NSComparisonResult result = [sd.suomi compare:searchText options:
    (NSCaseInsensitiveSearch|NSDiacriticInsensitiveSearch) range:NSMakeRange(0,
    [searchText length])];
              if (result == NSOrderedSame)
              {
                     [self.filteredListContent addObject:sd];
              }
```

This is the actual search function. When user types something with keyboard, that string is compared to dictionary words. If a searched word is found, it is passed to filteredListContent array and the content of that array is shown in search results table view. NSMakeRange makes the search to start from the first letter of every word. /6/

```
//Function used when user chooses a dictionary word.
```

```
- (void)tableView:(UITableView *)tv didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if(SdController == nil)
        SdController = [[SanaDataController alloc] initWithNibName:@"SanaData"
        bundle:nil];
        SanaDetail *sd = nil;
        if (tv == self.tableView) {
         SdController.sanaDetail = [self.ListaSanat objectAtIndex:indexPath.row];
    }
     else {
        SdController.sanaDetail = [self.filteredListContent
        objectAtIndex:indexPath.row];
    }
     SdController.title = sd.suomi;
     //Makes the new table view visible (which includes details of the words)
     [[self navigationController] pushViewController:SdController animated:YES];
}
//Hides the search bar when 'cancel' is pressed
- (void)searchDisplayControllerDidEndSearch:(UISearchDisplayController *)SdController {
     self.searchDisplayController.searchBar.hidden = TRUE;
}
```

This is the final function called in my first table view. When user presses any word, the next controller (SanaDataController) will be called and a new table view will be shown. /7/

4.2.4 Grouped tableview

This table view will show all three values for a chosen word. These values are Finnish, Thai and pronunciation. This table does not need to have as many rows as the previous one, so it will look different (fig. 15).

Figure 15: Grouped table view to show translation pairs.

```
- (void)viewWillAppear:(BOOL)animated {
    self.title = sanaDetail.suomi;
    [tableView reloadData]; }

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 3;
}
```

The title shown in navigation bar will be given here. Also the number of needed cells is given. /4/

```
- (UIView *)tableView:(UITableView *)tv viewForHeaderInSection:(NSInteger)section
{    NSString *sectionTitle;
    switch(section)
    {
      case 0: sectionTitle = @" Suomi"; break;
      case 1: sectionTitle = @"  Thai"; break;
      case 2: sectionTitle = @"Lausunta"; break;
    }
    //Create rectangular area above sections to hold the title.
   UILabel *label = [[[UILabel alloc] init] autorelease];
   label.frame = CGRectMake(123, 0, 300, 25);
   label.backgroundColor = [UIColor clearColor];
   label.textColor = [UIColor orangeColor];
   label.shadowColor = [UIColor whiteColor];
```

```
    label.shadowOffset = CGSizeMake(0.0, 0.5);
    label.font = [UIFont systemFontOfSize:20];
    label.text = sectionTitle;

    //Create header view and add a subview called label.
    UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 40)];
    [view autorelease];
    [view addSubview:label];
     return view;
}
```

The title for every cell is given here. ClearColor is used for transparent color to make the background image visible. Shadow for text is also possible and it can make text look better. /8/

```
-(NSInteger)tableView:(UITableView *)tv numberOfRowsInSection:(NSInteger)section {
     return 1;
}
```

All of the words should fit into one row so no need for multiple rows.

```
-(UITableViewCell *)tableView:(UITableView *)tv cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
     UITableViewCell *cell = [tableView
     dequeueReusableCellWithIdentifier:@"MyIdentifier"];
     cell = nil;
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
     reuseIdentifier:@"MyIdentifier"] autorelease];
     cell.backgroundColor = [UIColor colorWithRed:.300 green:.300 blue:.300 alpha:1];
     }
```

Creating cells. CGRectZero creates an optimal size cell automatically. Backgroundcolor is here set to gray.

```
     switch(indexPath.section)
     {
       case 0: cell.textLabel.text = sanaDetail.suomi; break;
       case 1: cell.textLabel.text = sanaDetail.thai; break;
       case 2: cell.textLabel.text = sanaDetail.lausunta; break;
     }
     return cell;
}
```

Getting right values for cells. /4/

4.3 Distribution

The application is uploaded to Apple Inc. App Store and can be downloaded to user's iPhone freely. When I finished my application, it was sent for review by Apple and it took five days to be verified and shared. For distribution user needs to have his/her Apple developer program fee paid and join Apple iTunes Connect service. Via that

service the applications are uploaded for verification.

When publishing an application, the following information is needed:

A) Application name: the official name which can not be changed unless the application is updated.

B) Description: This text will be shown to users who want to purchase or download your application.

C) Device requirements: On which platforms user wants the application to work.

D) Categories: The primary categories, for example 'utilities' or sports' to help users find the application from App Store.

E) Copyright: Copyright owner of the application.

F) Version number and SKU number: These numbers are added by developer to help identify the application.

G) Keywords: Word list for finding software in App Store.

I) URL and e-mail: These addresses are needed for users to find more information or get support with the application.

J) Ratings: Developer needs to define if the application includes any material that might not be suitable for all ages.

After submitting this information, there will be an upload page. In this page the real application executable is uploaded as well as possible screenshots and shortcut icons for iPhone and iPad. The next localization page is useful if application is implemented in several languages. The last information developer needs to give considers pricing of software. If the application is free of charge, there is no need to worry about this page.

4.4 Testing

While constructing the application it was tested numerous times in simulator. After seeing it working well in simulator, it was also tested in a real iPhone to see if it is fast enough. The application works well, so there is no need for optimization of the code at this point.

The real testing will happen when other users download this application and hopefully give their reviews of it. Later on there will be a survey on an internet forum to ask for more opinions. With help of other users, it is easier to make the application work better and make it include more features.

5. SUMMARY

Developing this application took a long time mostly due to my lack of mobile programming knowledge. I had several problems along the way, but managed to solve them by testing the code and reading plenty of information online. I learned a lot about mobile programming and I am satisfied with the result. This application can now be used as a base for new applications. In the future there will hopefully be new versions of this software and with it new useful properties.

6. REFERENCES

Online references

1 Picture taken from:
ASP.NET MVC QuickStart 1: Introducing ASP.NET MVC.
http://ludwigstuyck.wordpress.com/2009/06/10/asp-net-mvc-quickstart-1-introducing-asp-net-mvc/
Search date: December 4th 2009.

2 Picture taken from:
Command Line Shell For SQLite
http://www.sqlite.org/sqlite.html
Search date December 16th 2009.

3 Code partly copied from:
iPhone SDK Tutorial – Using SQL Lite Part I
http://www.iphonesdkarticles.com/2008/07/iphone-sdk-tutorial-using-sql-lite-part.html
Search date: December 20th 2009.

4 Code partly copied from:
iPhone SDK Tutorial – Using SQL Lite Part II
http://www.iphonesdkarticles.com/2008/07/iphone-sdk-tutorial-using-sql-

lite-part_21.html
Search date: December 20th 2009.

5 Code partly copied from:
iPhone Programming Tutorial – Populating UITableView With An NSArray
http://icodeblog.com/2008/08/08/iphone-programming-tutorial-populating-uitableview-with-an-nsarray/
Search date: December 21st 2009.

6 Code partly copied from:
Search the content of a UITableView in iphone
http://edumobile.org/iphone/iphone-programming-tutorials/search-the-content-of-a-uitableview-in-iphone/
Search date: January 15th 2010.

7 Code partly copied from:
UITableview – Searching table view
http://www.iphonesdkarticles.com/2009/01/uitableview-searching-table-view.html
Search date: January 15th 2010.

8 Code partly copied from:
Changing Background Color and Section Header Text Color in a Grouped-style UITableView
http://undefinedvalue.com/2009/08/25/changing-background-color-and-section-header-text-color-grouped-style-uitableview
Search date: February 25th 2010.