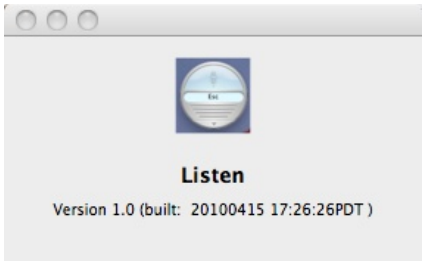
LISTEN TUTORIAL

USING THE SPEECH RECOGNIZER & SPEECH SYNTHESIZER

This project is a little bit of fun. As you know there's a lot of technology in MacOSX and it will come as no surprise that the system has both a speech recognizer (for converting sound to strings) and a synthesizer for converting strings to sound.



And because I always like to throw a 'bonus' item into every project, I've also added a custom build step to the project so that the 'About Box' displays the date and time of the build. Every time you build, the About Box is updated.

A word about the recognition. I found it difficult to get the recognizer to understand my Scottish accent. You may have to be quite determined to get this application to perform. You also have to press 'esc' to speak to the machine.

You can download the code from: **http://clanmills.com/files/Listen.zip**

The program has a very simply User interface - simply a couple of buttons, a customized 'About Box'. The system provides the speech recognition window when the recognizer is listening. Our code will make that appear and disappear appropriately.

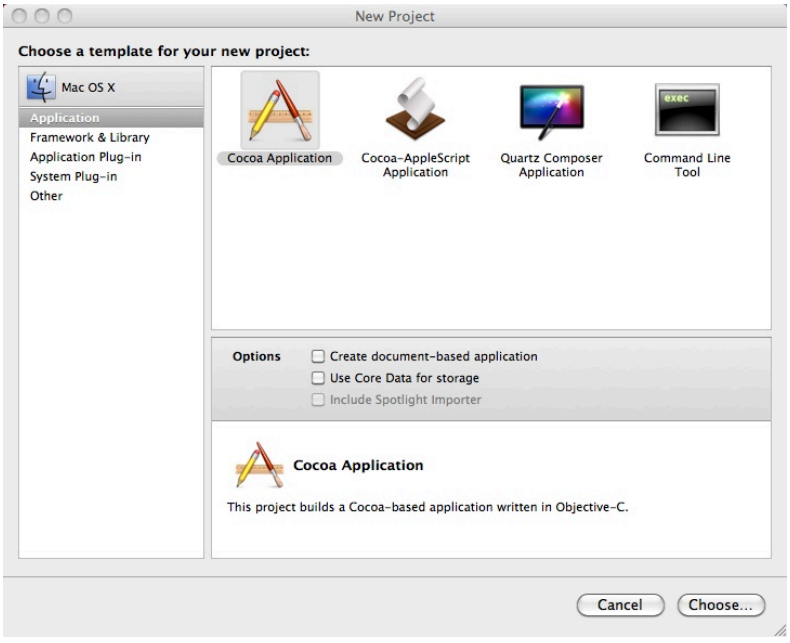
ACKNOWLEDGEMENT

I wish to acknowledge the contribution of 'routers' on http://www.cocoaforum.com This person appeared on the forum offering code and asking for assistance. Thank you routers for introducing this subject to the forum and for the code that you provided.

RECIPE

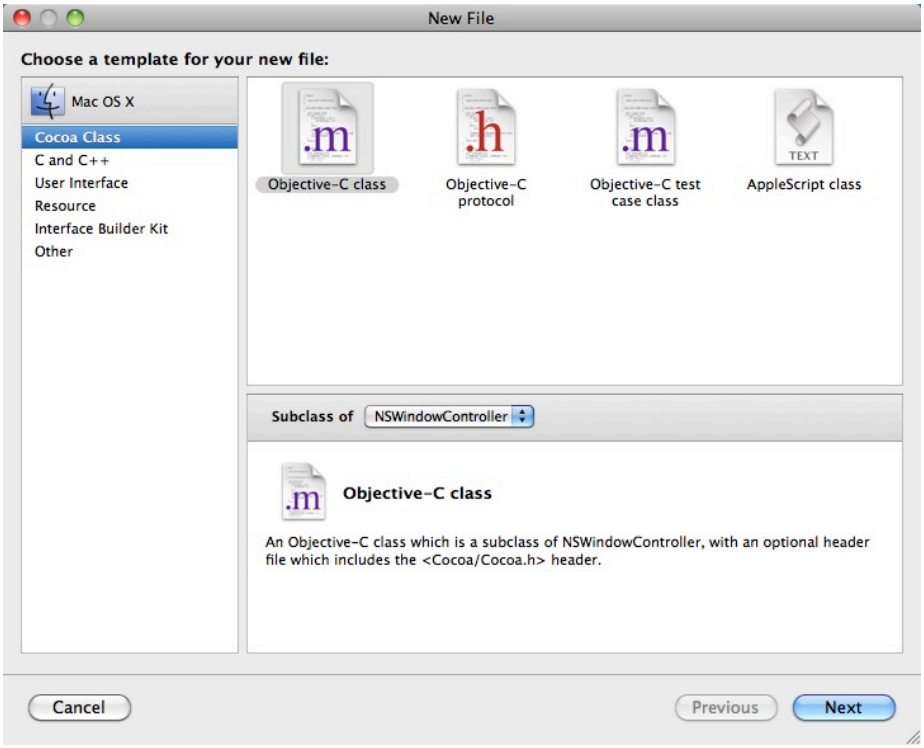
1) Get the Wizard to generate a project for Listen

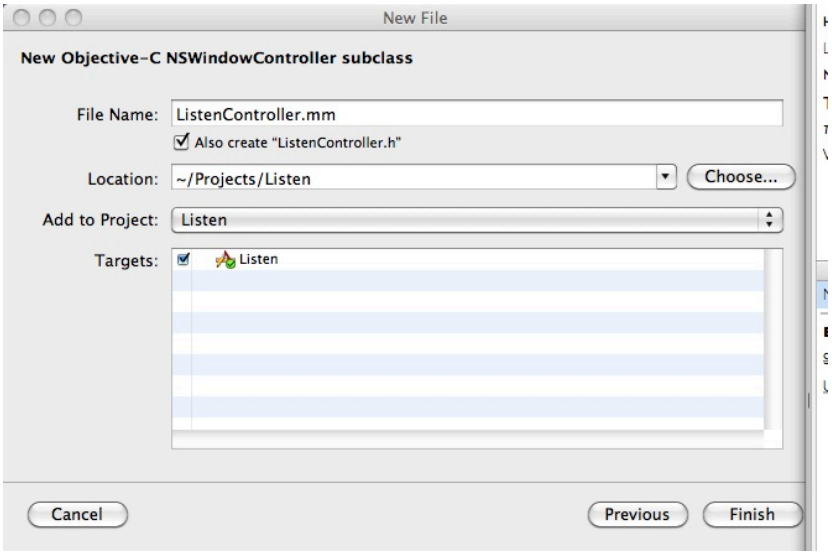
Start with a new Cocoa Application



Rename main.m to main.mm, and ListenAppDelegate.m to ListenAppDelegate.mm (this makes it Obj/C++ instead of Obj/C)

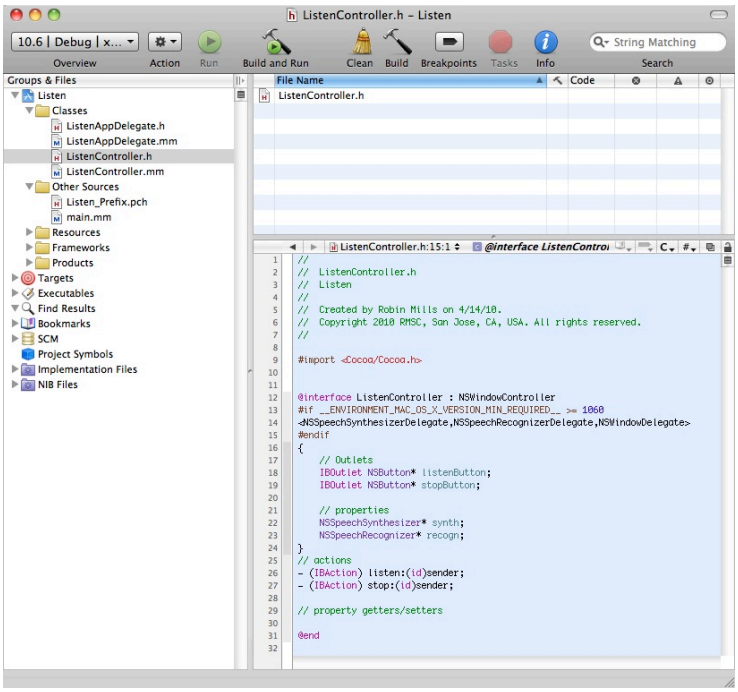
2) Add a new class ListenController based on NSWindowController





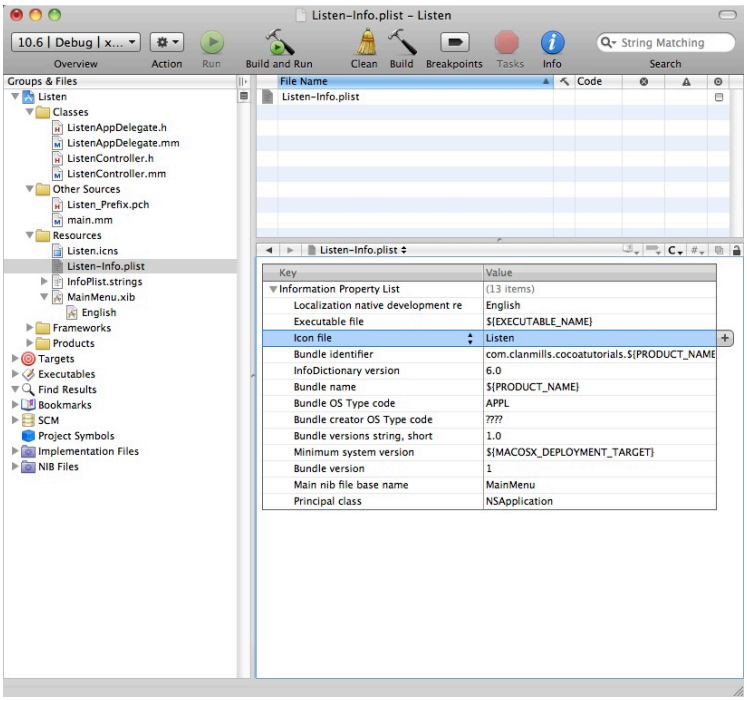
3) Add the code from the web site to ListenController.h and ListenController.mm

We'll discuss the code later. XCode should should look like this:



4) Modify the Resources

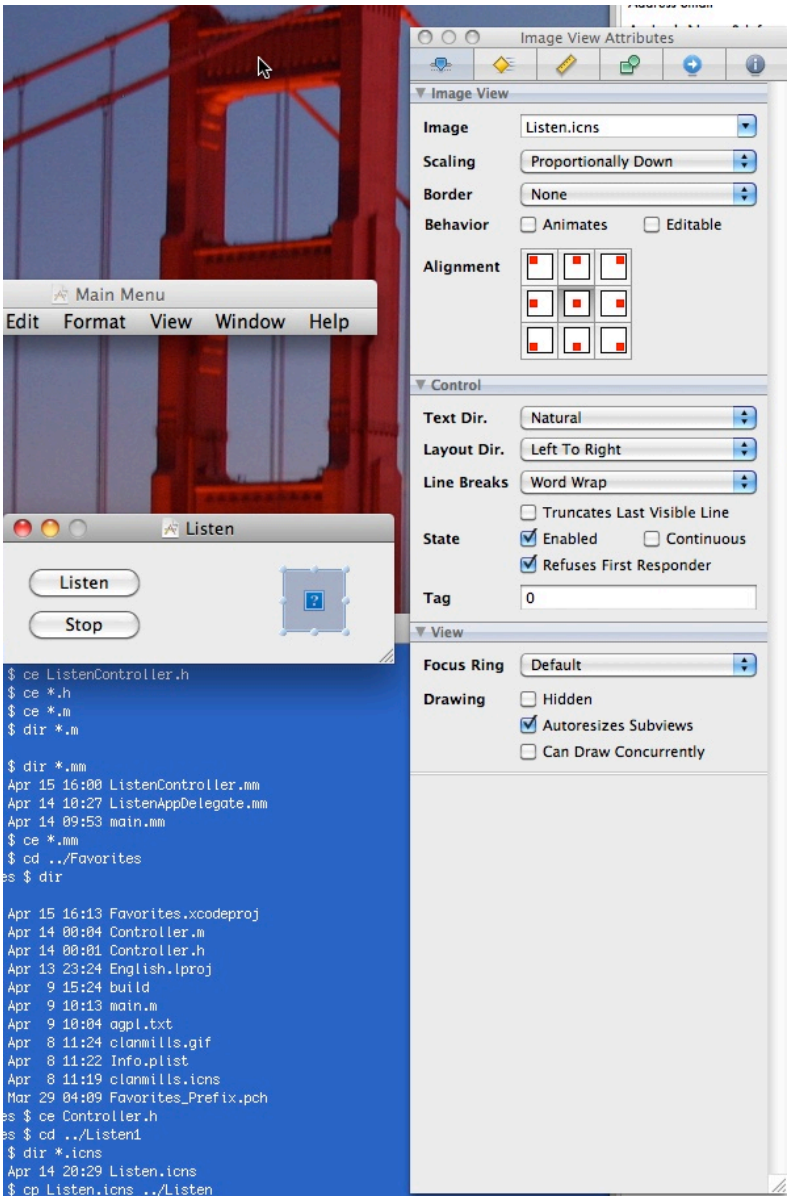
Copy the file Listen.icns in the Listen directory.
Add Listen.icns to the Resources in XCode.
Edit Listen-Info.plist and set Icon file to be Listen (not Listen.icns).
I also modified the Bundle identifier just for fun.



INTERFACE BUILDER

1) Open MainMenu.xib in Interface Builder (double-click the file in XCode)

Layout out the UI as shown. You should use an NSImageCell for the graphic, and assign the Image View Attributes as shown (notice the consistency: in Listen-Info.plist the icon file does not have an extension, in Interface Builder it does!)



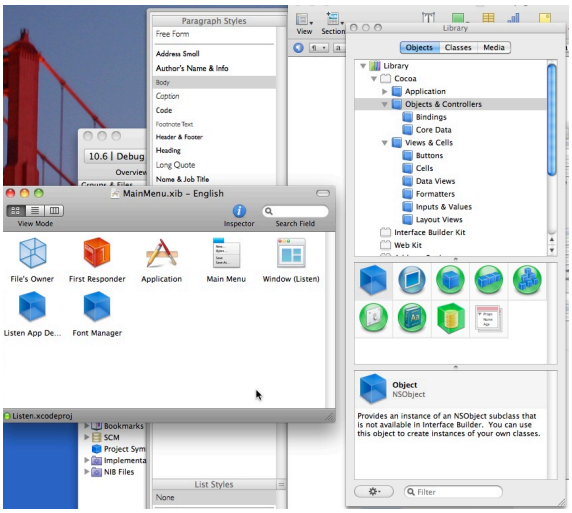
Save the files in Interface Builder. Build and Run in XCode. You should see this:

The Listen and Stop buttons do not yet have any functionality.

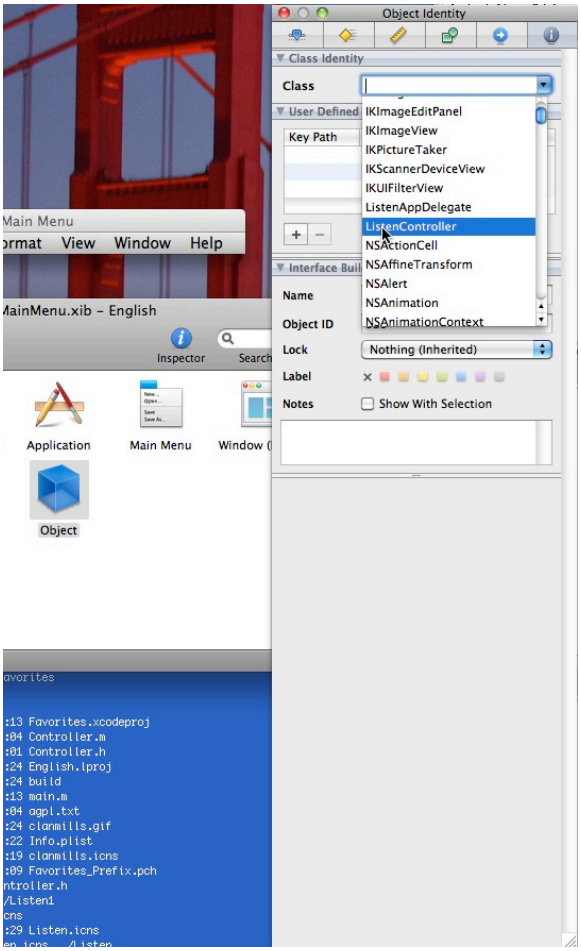
2) Export ListenContoller.h from XCode to Interface Builder

1) Drag'n'Drop ListenController.h from XCode onto MainMenu.xib in Interface Builder (no visual feedback)

2) Drag'n'Drop an NSObject (a cube) from the Library/Objects into MainMenu.xib in Interface Builder.



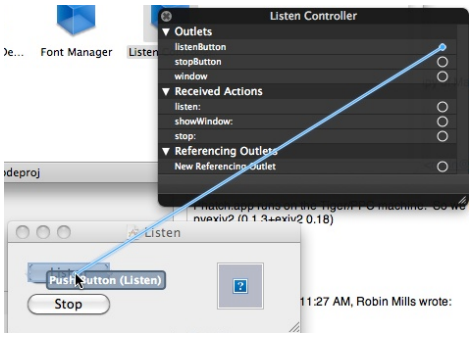
3) Select the new 'cube' called 'Object' and Assign it to the class ListenController.



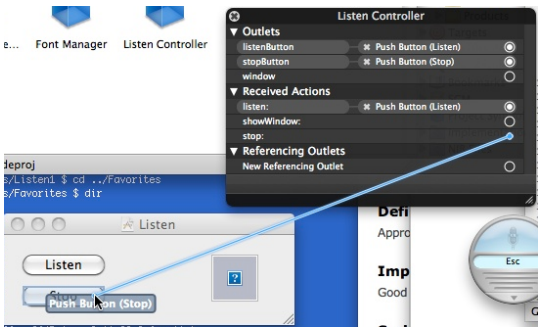
The cube called 'Object' will be renamed as 'Listen Controller' (with a gratuitous space). You can now right-click that object to gain access to the Outlets and Actions you defined in ListenController.h

4) Connect the Outlets and Actions

Remember Outlet = Control (so listenButton = Something you can see)



Remember Action = Event (so you'll recieve the Stop event on method stop



5) Save, Build and Run

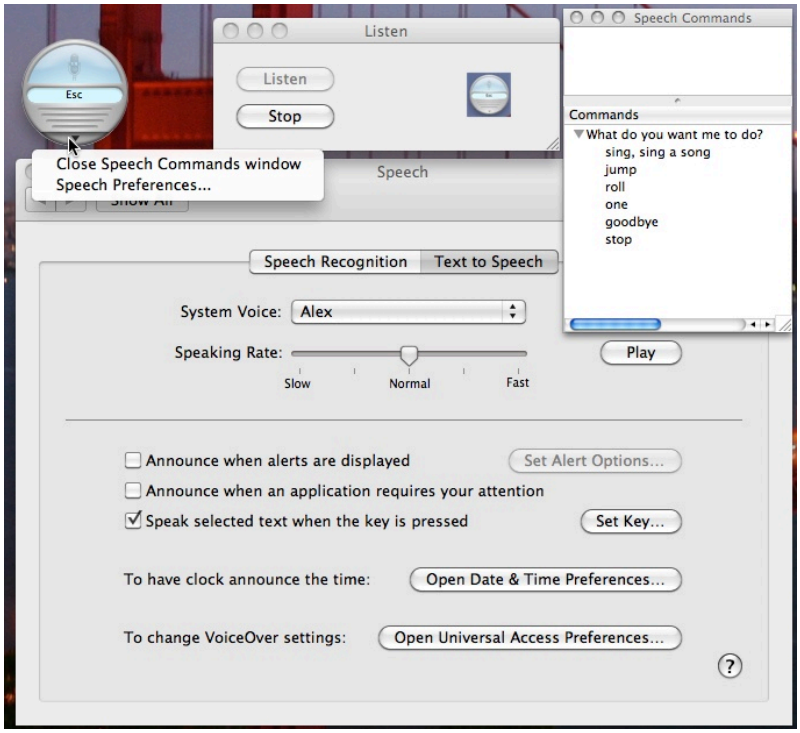
RUNNING THE PROGRAM

When the program runs, you are presented with the following dialog box:

When you press 'Listen" a couple of things happen. Firstly, the "Listen" button is disabled (grayed out) and the "Stop" button is enabled. You see the systems Voice Recognizer:

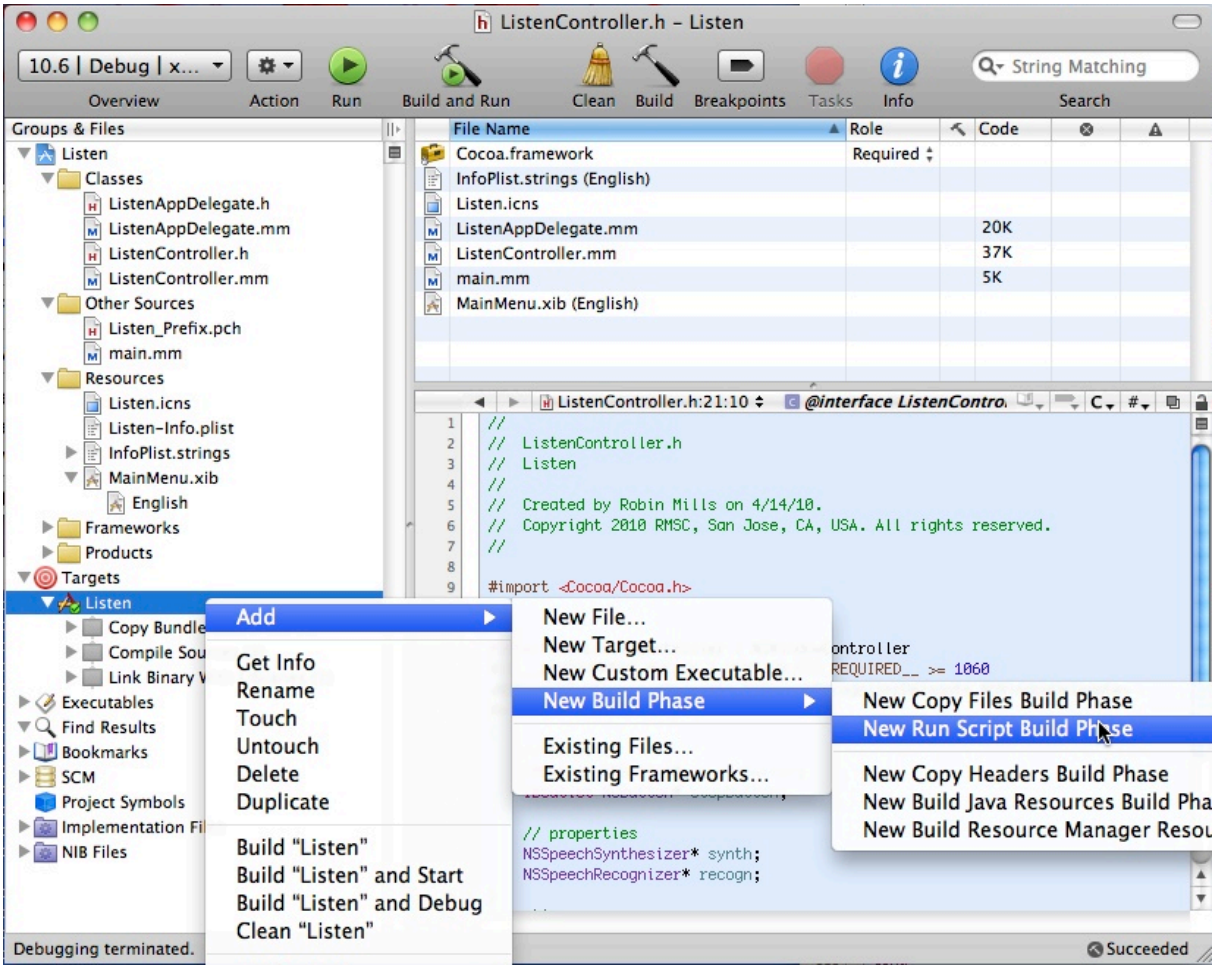
You have to press "Esc" to speak to the machine. The key words are "sing, sing a song", "jump", "roll", "stop" and "goodbye". There is an optional additional window which displays the current vocabulary. I found the recognizer was not good with my Scottish accent and that's why I added the "Stop" button to the UI.

The Speech Commands can be optionally displayed. System Preferences "Speech" is available to change the default voice and other settings.



ADDING A CUSTOM BUILD STEP IN XCODE

This is not necessary at this time. Skip this if you wish.

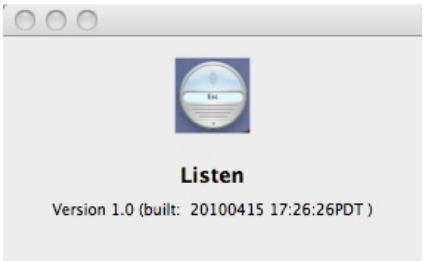


Edit the 'Run Script'. Use the shell /bin/bash. You'll want to move the 'Run Script' in XCode to be done before you

```
buildInfo=`/usr/bin/python -c "import time; print \"'built: \",time.strftime('%Y%m%d %H:%M:%S%Z', time.localtime()),\"'\""`
/usr/libexec/PlistBuddy -c "Set :CFBundleVersion '$buildInfo'" Listen-Info.plist
```

For build-engineering types, you might wonder. Hey, what happens if that file is under source code control. Everytime we build, we'll have to submit the file. I can't do that Fix it! Well, you can script your way around that. For example you could revert the file in a later step in the build. You could copy/restore it. Lots of possible tricks. Let's not discuss that. XCode's OK.

Clean, Build and Run Listen. Inspect the 'About Box'. Do this a couple of times to be sure the magic works.



DISCUSSION ABOUT LEARNING COCOA AND OBJ/C

I'm not sure why learning Cocoa is challenging. Some people will find it very easy and intuitive, however most do not. I think there are several different obstacles facing a new user the these include:

1) The terminology being used. Obj/C introduces unfamiliar term such as delegates, outlets, actions.

2) Interface Builder
   This is a strange beast.

3) The Memory Manager
   Prior to Obj/C 2.0, there was no garbage collector. All objects are reference counted. However it's often unclear to new users when to retain or release an object.

4) The Obj/C language is of course C. However there syntax has been extended to deal with messages.
   I think this is simply something the new user has to learn. In time the syntax will seem invisible.

I'm going to discuss these challenges in this tutorial.

Additionally, however there are other matters which are quite daunting for the new users. Although I recognize the difficulties presented, I'm not sure there's any simple solution. It takes time to learn anything. However I also believe that the rewards provided by Cocoa and Obj/C are worth the effort required in learning.

5) The Frameworks and Design Patterns being used.
   Without doubt, Cocoa uses some best of practice patterns which have to be learned and appreciated.
   CoreData and Binding are very powerful mechanisms. With great power comes great responsibility, grasshopper.

6) Inter object communications
   As with all software frameworks, there are many possible ways to do things. For example it is possible for objects to communicate using a 'delegate' method. Or the objects may have pointers to each other in their properties (or the properties of properties). Or they could use Notifications.

   I'm not going to tackle this subject at this time.

7) The size of the product. It is rather staggering in its scope. Even the Foundation class for Strings takes a while to absorb.
   Once more, I think time and practice with the frameworks are required to reach a level of comfort.

8) There aren't a lot of books available about Obj/C and Cocoa.
   Aaron Hillegass' book is very good and covers many things. Aaron's a great teacher and writer. And of course Aaron and his publishers have to be selective in the amount of material they provide. I'm trying to extend that knowledge base with my tutorials.

1) Let's talk about delegates, outlets, actions and properties

A **Delegate** in Cocoa is called a 'callback' in other architectures.

When you create an instance of the Speech Recognizer. Of course it has properties and methods. Properties include an NSString displayedCommandsTitle. When set, the recognizer provides an additional window to show its title and expected commands. Methods include startListening and stopListening - no explanation is required. And it has the all important setDelegate method.

When you use setDelegate you pass as id (NSObject*). Any object can be a delegate. The object for whom an object is a delegate will be called on defined callbacks. That's it. A delegate is a callback object.

Curiously in Obj/C, an object can be a delegate for any number of objects. And an object can be a delegate for more than one class of object. Starting with XCode 3,.2, there is additional syntax on the declaration of an object to say which delegate classes are implemented. That syntax is:

```
@interface MyClass : SuperClass <SomeDelegate [,...]>
```

I, personally choose to express this with a conditional compilation to enable the code to be build with XCode 3. (See the tutorial Favorites.pdf for a longer explanation).

```
@interface ListenController : NSWindowController
#if __ENVIRONMENT_MAC_OS_X_VERSION_MIN_REQUIRED__ >= 1060
<NSSpeechSynthesizerDelegate,NSSpeechRecognizerDelegate,NSWindowDelegate>
#endif
```

An **Outlet** in Cocoa is called a 'control' in other architectures.

When I declare an interface (a new class of object) in Cocoa, I set out the code as follows:

```
@interface ListenController : NSWindowController
#if __ENVIRONMENT_MAC_OS_X_VERSION_MIN_REQUIRED__ >= 1060
<NSSpeechSynthesizerDelegate,NSSpeechRecognizerDelegate,NSWindowDelegate>
#endif
{         // Outlets
          IBOutlet NSButton* listenButton;
          IBOutlet NSButton* stopButton;
          // properties
          NSSpeechSynthesizer* synth;
          NSSpeechRecognizer* recogn;
}
// actions
- (IBAction) listen:(id)sender;
- (IBAction) stop:(id)sender;

// property getters/setters
// methods
@end
```

An Outlet is simply a control in the UI. If you use a textField and wish to manipulate it at run time, you should declare an IBOutlet NSTextField* textFieldName. When your UI is create during program initialization, your IBOutlet variables will be initialized. If you wished to enable/disable the control at run time, you would send a message with [textField setEnabled:YES]; (or NO to disable).

An **Action** in Cocoa is called an 'event' in other architectures. An action is a method provided by your object which will be called by the control. In other words - it's an event.

Notice however that the syntax of IBOutlet and - (IBAction) are remarkably different. I believe this is because (IBAction) is a macro for (void). IBOutlet is a macro for ** nothing at all **. When Interface Builder reads .h files, he is looking for IBOutlet and (IBAction) to locate your Outlets and Actions. However in C, these things are almost invisible.

A **Property** in Cocoa is called a member variable in other architectures. There isn't much to be said about this. Obj/C 2.0 did bring new syntax for the compiler to create getters and setters. Additionally the syntax instance.property (both for get and set) are now provided in Obj/C. No need to use x = [instance property]; and [instance setProperty:value] messages.

2) Interface Builder

Interface Builder feels like magic. As I have explained above, Interface Builder is able to preprocess object classes (@interface ... @end declarations in Obj/C. In Aaron Hillegass book, he often discusses objects being stored in the NIB. It may seem as though the object is stored in the Interface Builder - however this is not the case. Interface Builder knows the names of classes. At run time when the NIB is loaded the object is constructed in memory by the code in XCode (the Obj/C), and of course the IBOutlets are constructed. When this has been successfully done, the run-time system calls awakeFromNib. So run-time initialization takes place in three stages.

1) Your object's init: method is run of course.

2) The runtime system does some magic with the IBOutlet objects.

3) Your awakeFromNib: method is called to tell you that you are alive.

At run time, the controls in the UI receive events from the operating system and the window system. The UI objects call your IBActions appropriately.

In order for Interface Builder to generate the appropriate code, you have to carry out the following steps in Interface Builder:

1) Tell him to read a header file (well, in fact he parses any @interface you present to him and remembers the object classes).

2) You can create an object (with the Library). The simplest controller object is NSObject and this is used by Listen.

3) You assign an object the property of a class name in Interface Builder. Now when you right-click on the object, you'll be presented with the Outlets and Actions and you can drag a connection with the mouse between an Outlet or Action and its corresponding visual element in the UI. For example a button, menu item or whatever.

Interface Builder in XCode 3.x is very much better than it was in the 'good old days'. The right-click and drag is much easier and more intuitive than the ctrl-drag that once was. Additionally, Interface Builder does not need to be instructed to re-read header files when they are changed. XCode 3.x usually seems to understand when Interface Builder files have been modified.

None the less, Interface Builder is an intimidating experience for new users. It offers a bewildering array of options, the purpose of which is totally incomprehensible to someone beginning to learn the system.

3) Memory Management

Memory Management is done by reference counting. Here's an article on the forum about this memory management. http://www.cocoaforum.com/2009/06/16/help-with-memory-management/

In Listen, I've added reference counting reporting around the allocation and releasing of the variable recogn.

When the garbage collector is not being used, you should remember the words of the Cocoa Country'n'Western Song: "Please release me if you m'alloc'ed, and don't retain me any more" If you malloc, you should release. If you retain, you should release. Never call dealloc. Never use an object after you have released it and assign the variable to nil.

You have to be on your guard however as it's not always clear who is responsible. I will write a tutorial on this subject one day.

4) Obj/C Syntax

The syntax of Obj/C often doesn't feel good. I remarked about (under Outlet and Action) about the lack of consistency between two elements of syntax. I am equally squeamish about the syntax of messages. In 'C' (and C++) the signature of a function are parameters. Since Ansi-C (and forever in C++), prototypes have been available to enable the compiler to test your calling arguments against the signature defined in the header file. A function is 'C' has the syntax:

```
return-type methodName(type1 arg1, type2 arg2 ...) { body }
```

A method in Obj/C has the syntax:

```
[+|-] // let's not discuss this because I like this syntax for a class and instance method/property
(returntype) methodName : (type1) arg1 argName2: (type2) arg2 .... { body }
```

The signatures simply all reversed to me. The first argument is unnamed. How odd! That's the way it is.

Additionally when you send a message (call a method), you have to provide the argNames in the correct order and of course that have to be spelt correctly. The system in turn converts them to:

extern "C" id obj_msgSend (id self, SEL op, ...);

And lastly there is the confusing matter of the selector which is the C type SEL. This is an integer. Every methodName and argumentName is stored in the runtime system exactly once. All references to a name are compiled into the code as constant integers. These integers cal be accessed using another alien construction @selector(name).

Obj/C of course doesn't have operator overloading. So + cannot be overloaded for a string to mean concatenation. This results in more ponderous code, however this doesn't seem very important to me.

Obj/C (and Obj/C++) can of course be linked with C and C++ libraries. There is no problem with this. It's also possible for C (and C++) programs to call Obj/C objects using the extern "C" signature I listed above. Of course your C code will have to know the id of the object to be called and the SEL for each name. I've never wanted to do this. Maybe one day, probably not.

A few kind words for the design of Obj/C, Cocoa and Interface Builder

This system delivers amazing results in a very elegant way. All who have mastered this have come to love and admire it. So while the effort may be considerable, the rewards are a valuable return on investment.

The Obj/C object model is much more flexible than C++. The difference is in the 'late binding' of the system. With C++, the compiler insists on knowing everything about your intention. Of course he produces very fast code as there is very little effort is required at run time to know the address of a method or member variable.

Obj/C on the other hand is much more like JavaScript. Objects can add and remove methods and properties at run-time. Of course it does this with some performance cost. However for most UI programming, I don't think performance is of great importance. If the UI is going to do some 'heavy number crunching", this is probably going to be carried out by a core library which is probably written in C++. Of course system responsiveness is very important to the user, however the UI elements (buttons and other widgets) are not the bottle neck..

And to be fair to Interface Builder, it's a 'heavy hitter'. It has an amazing amount of functionality built into a GUI. It has to be in the GUI because the NIB file format is a secret - so Interface Builder is the only way to create a NIB. XCode 3.0 brought the .xib format is an XML formatted version of the nib data. I've only glanced at that. I don't think many people would want to generate .xib files with other tools - but you never know.

However I keep my greatest praise for the functionality of the Cocoa Frameworks. Apple have done a great job. There's a formidable amount of functionality available in this system. The UI, Image manipulation (such as PDF), Core Graphics, Core Data, File and Network I/O, speech recognition and synthesis and lots more. The software is very capable. And of course it is 'C' so you can call open source libraries if you wish. Apple (and the Next Step) team have done a great job with this system.

I am in awe of Obj/C, Cocoa, XCode and Interface Builder. I am also in awe of DevStudio and Eclipse.

CODE DISCUSSION

The code in the application is very simple. I think you'll be able to understand it almost at a glance.

1) init: dealloc: and awakeFromNib:

This is very straightforward 'everyday' code. init ensures that recogn and synth are not undefined. dealloc releases resources.

I've also called a little "C" function dictDump from init. This is for the fun of seeing that our custom build step has indeed modified Listen-Info.plist on every build.

The recogn variable is not alloced until we wish to use him. This happens in listen: and he is released in stop:

awakeFromNib: also enables our "Listen" button and disables the "Stop" button. Don't do this in init: as listenButton and stopButton have not yet been constructed by the run-time system. Wait until awakeFromNib: to interact with your Outlets.

I also assign our object to be the delegate (callback) for the Window object and the NSSpeechSynthesizer. When I alloc the NSSpeachRecognizer object is listen:, I make our object a delegate (callback) of recogn. Objects can be delegates (callbacks) of any number of other objects (which can be of any class).

2) windowShouldClose:

I almost always define this delegate to return YES. When the window is closed, the application quits.

3) listen: and stop:

I think these methods are rather obvious.

I've complicated them a little by adding NSLog messages about the reference counting of the recogn object. You'll notice that the recognition disappears from the display when its retainCount == 2. Why? Well, we release it (so it retainCount will then be 1). Never inspect an object after you have released it (well you can look with the debugger). Presumably there's some code further down the software stack which knows about this object (otherwise the recognizer couldn't work). So he'll have a reference. There must be code somewhere that says "OK, retainCount is 1 and that's us, so let's clean up".

Notice that if you comment the [recogn stopListening], the reference count remains at 2 and the recognizer does not disappear from the display.

CHALLENGES

I haven't thought of any ways to extend this program. I personally found it so difficult to get the recognizer to understand me that I didn't feel much enthusiasm to explore further. Additionally, having to use the 'Esc' to enable the recognizer is a very unpleasant User Experience.

If you have great ideas and make more discoveries about this, please email and I'll be happy to update the tutorial and acknowledge your contribution.

LICENSE

Robin Mills
Software Engineer

Windows/Mac/Linux
C++, Web, JavaScript, UI

400 N First St #311
San Jose, CA 95112

T (408) 288 7673
C (408) 394 1534
robin@clanmills.com

http://clanmills.com

REVISION HISTORY

20100416	Initial version.