Tyson Hubbard
CS 370

iOS Development

Talking with my father about his trouble hearing people while talking on the phone led to the idea of creating a mobile phone app to act as a 'hearing aid,' not just in phone calls, but any other media run over the device. Provided the device has the right hardware, this hearing aid app could potentially act as a hearing aid in the more general sense, to help people in hearing the sounds around them.

In its simplest form, this application would be of great benefit to people who have trouble doing anything--especially anything official or professional--over the phone due to trouble hearing. If I can take the idea further, and make an app that is more of a general-purpose hearing aid, then my application could be much more useful for some people in their daily lives. Some people, like my father, are only starting to have trouble hearing people over the phone. In fact, he says that for the most part, it is mostly women whom he has trouble hearing (which stands to reason, since people start to lose the ability to hear higher frequencies first). Of course, some people have much more trouble hearing, and it affects their daily lives. If I could turn a smartphone into a general hearing aid, it could provide improved hearing to many, many people for whom in-ear hearing aids are prohibitively expensive.

I have read that those in-ear devices can cost as much $5,000. Although smartphones are not necessarily cheap, they do not cost $5,000, and they are quickly pervading the market. If you have trouble hearing, you may already have a smartphone, in which case, all it will cost to improve your hearing is a small fee in the app store. If you don't have a smartphone, it is very likely that you can get one for, say, $100. Either way, my app would make it possible for just about anyone to improve their hearing for no more than a little over $100. Based on this line of reasoning, I believe that this app could be both very helpful for perhaps millions of people, and (therefore) very profitable for me.

The first decision to make about developing this mobile app was which mobile platform to use. I only considered Android and iOS, since they are the two biggest. In many ways, I prefer Android for personal use, but for a number of reasons, I ultimately chose iOS. First, since Android is an operating system for mobile devices, and the manufacturers of those devices are diverse, the hardware varies pretty widely, and this leads to the need for testing on multiple hardware sets. I have a feeling I will have to write code that interacts directly with the audio hardware, hence, I prefer working with a hardware set that is more consistent.

Having decided to develop my app for the iphone, the next goal was to learn Objective C and become familiar with XCode. I had read that Stanford offers a free course on iOS development, and after reviewing a little of the material, I decided it would be a very good way to learn. Through the development of a graphing calculator and then an app to access pictures through Flickr, the Stanford course teaches the fundamentals of iOS development, including Objective C and XCode.

Objective C

Object Orientation

All the objects created in the assignments from the Stanford course inherit from NSObject, a class that is part of Apple's Foundation Framework. This is part of the objective side of Objective C. All objects that are subclasses of NSObject inherit a whole host of methods for instantiation, allocation, deallocation, etc. This includes methods for introspection, for example asking a dynamically typed object what class it is; and methods for bug testing and memory management.

Dot Notation with @property and @synthesize

One very useful feature that is a more recent addition to Objective C is @property and the use of @synthesize for @property declarations. An @property is used with objects in order to provide information about that object to other objects. This allows objects to protect their internal information, while allowing other objects to query for that information. Then, @synthesize is used to auto-generate both a setter and a getter method for the object. For instance, if one designates a property as @property origin for a graph, then typing @synthesize generates a setter called setOrigin and a getter called origin, tying to some internal variable called origin. The programmer can also override either the setter or the getter by declaring his own, with any extra effects he may want. Among other modifications, an @property can also be tagged as read-only, preventing the compiler from generating a setter method. This combination is very useful, and it also allows for the use of dot notation for getter methods, which is more brief and often easier to read than the previously more common notation involving enclosing the getter and the object in question in brackets (e.g. "graph.origin" rather than "[graph origin]"). One of the most useful things about dot notation is chaining dotted getter methods together, as in "graph.origin.x" or "graph.origin.y." This is much easier to read, and I would argue it is faster to type, as well.

Memory Management

Memory management in Objective C is actually relatively simple in a lot of ways. The most important rule for memory management in Objective C is, "if you own it, release it." If an object is "alloc"ed, copied, or declared with "new," then the instance of the particular class in which that object is allocated, owns that object. Therefore, the simple rule of thumb is to make sure, to match all "allocs," or "copies," or "new"s with a release of that object when you are done with it. This even works with view classes, and once an instance of that class has been passed to its controller, the view object can be released, because the controller has its copy of that view object. The controller uses the "retain" command, which creates a copy of the view object that was passed to it. Retain is what allows the programmer to pass an object and then release it, by preventing that object from getting de-allocated out from under the class it has been passed to.

XCode

MVC

One of the first principles to learn when it comes to iOS development is the Model-View-Controller division of software components. For example, the work necessary for the calculator is divided into the calculator brain--the model; the calculator view controller--

the controller; and by the end of the project, the calculator has two views: the touchscreen calculator view and the graph view for displaying graphs of equations entered into the calculator. The work of performing the mathematical functions requested by the users is done by the calculator brain; the controller manages communication between the views and the model, and it also determines what view should be displayed at any given time. This division of work among the parts necessary to run any application serves to minimize guesswork when it comes to debugging, because if, for instance, the view is not displaying something properly, then either there is a problem in the view, or in the way the controller communicates with the view. The programmer does not have to look anywhere else. It is important to maintain this philosophy of organization throughout the project, and one main tenet of that philosophy is that the model never communicates directly with the view. Both the model and the view communicate with the controller, and never each other.

Compared to Other IDE's

XCode provides the standard color coding and auto-tabbing of code that one would expect from any modern IDE, but the built-in debugging features are also very advanced, allowing a user to chase down a bug through the series of method calls that led to a halt, for example. The console is very easy to use, and one of my favorite features is a 'fix it automatically' option for simple, minor errors. Very often, this option offers exactly the desired fix.

Interface Builder

Although it seems that Apple is very fussy when it comes to defining what is allowed in developing for iOS, the system they have set up is actually quite powerful. The Interface Builder that is built in to XCode allows for the creation of a very customized and polished interface, with a simple drag-and-drop structure for placing touchscreen buttons and sliders (etc.) and assigning them to functions. One is not limited, however, to creating views in this manner. For instance, in the calculator application from the Stanford course, at first the graph is generated with the drag-and-drop interface, and this view includes a pair of zoom buttons. Later, the student is required to generate a view without this interface, and use the gesture recognizer class to pan around the graph view and for zooming in and out. The gesture recognizer class is provided by Apple for easy interpreting of a user's touchscreen gestures, and it is a very simple way to provide the proper screen response to a user's gesture input.

Calculator

Problem Description and Stages

The first four assignments in the Stanford iOS development course involve building up a graphing calculator. At first, the problem is relatively simple: produce a small application that displays a classic calculator, with a few minor functions. Assignment 2 adds the challenge of building in an equation evaluator, that can substitute values for variables in expressions and return a solution to the expression. Assignment 3 involves using this expression evaluator to produce a graph of the function, by substituting a whole series of values into an expression and evaluating. Finally, assignment 4 is to make the calculator a universal iOS app, by extending functionality to the iPad.

The first assignment is very much a walkthrough, and there is not so much code that has to be written from scratch. The student follows the walkthrough to generate much of

the calculator, and then takes similar steps to add a few more operations and some memory functions to the calculator. The main goal here is to get used to XCode, the Interface Builder and Objective C syntax.

Components

All it takes to make the calculator interface is to drag-and-drop all the buttons needed into place, along with a display panel for the numeric calculator interface. The buttons all connect to methods in the CalculatorViewController class. Number buttons, along with the decimal point button, are connected to the digitPressed method, operation buttons are connected to the operationPressed method. One of the first puzzles in the first assignment is how to disallow the user from entering multiple decimal points as they enter a number, producing invalid input. I chose to add a simple boolean variable that gets set to true the first time the user presses the decimal point, and prevent any further input through the decimal point button until after an operation has been pressed, resetting the display for number input.

At this point in the assignment, the model is the class CalculatorBrain, the controller is just CalculatorViewController, and the view is just the CalculatorViewController.xib file, which is actually an xml file that contains all the necessary markup to generate the view that was created using the Interface Builder drag-and-drop interface. This is a description of the Model-Controller-View organization of the calculator at this early stage.

In Assignment 2, the MVC arrangement remains the same, and the student is asked to add a few variable buttons, and an evaluate button. This requires building an array that will contain the series of numbers and operations and variables that the user enters, and including a method that can evaluate these expression arrays and return a number. In my testing, I was challenged by several puzzles about how to display the expression as the user was in the process of entering it, and I was ultimately satisfied with the way I was able to make it function.

Assignment 3 requires the student to add graphing functionality to the calculator and replace the evaluate button with a graph button, which produces a graph of the expression that the user has entered. This meant reducing to one variable button, namely “x,” and evaluating the expression at a whole series of points in order to plot the resulting series of “y” values. This graph is displayed in a graph view, which means that since there is a new view, there must be a controller for that view. Technically there should be a model to manage the details behind the scenes, too, but since the behind-the-scenes data is minimal (essentially just the expression) this view’s model involves piggy-backing on the main model for the whole calculator.

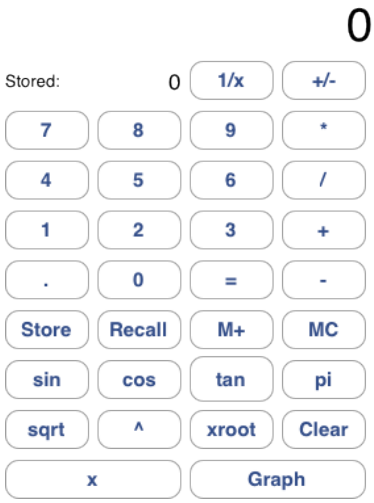
The fourth assignment just involves implementing a few extra functions necessary for upgrading the app to be a universal iOS app that can also run on an iPad, and going through the upgrade process in XCode. This process helps to illuminate the necessity for a view controller for every view. Since the iPad has a bigger screen, there are two main view modes that are used to take advantage of the extra space: a split view and a popover view. For easy implementation of these forms, Apple provides both splitViewController and popOverController, and in order to use this functionality, one passes his own view controllers to these controllers. This “controllers of controllers” approach is part of the MVC programming philosophy.

When the iPad is oriented horizontally, the split view is used and the calculator takes up about fifteen percent of the screen and sits on the left, with the graph occupying the rest of the screen. In vertical orientation, the graph takes up the entire screen, and the calculator can pop

over a small portion of the screen from a button at the top of the screen. In this pop over view, the calculator is essentially the same size as it originally appears on the iPhone.

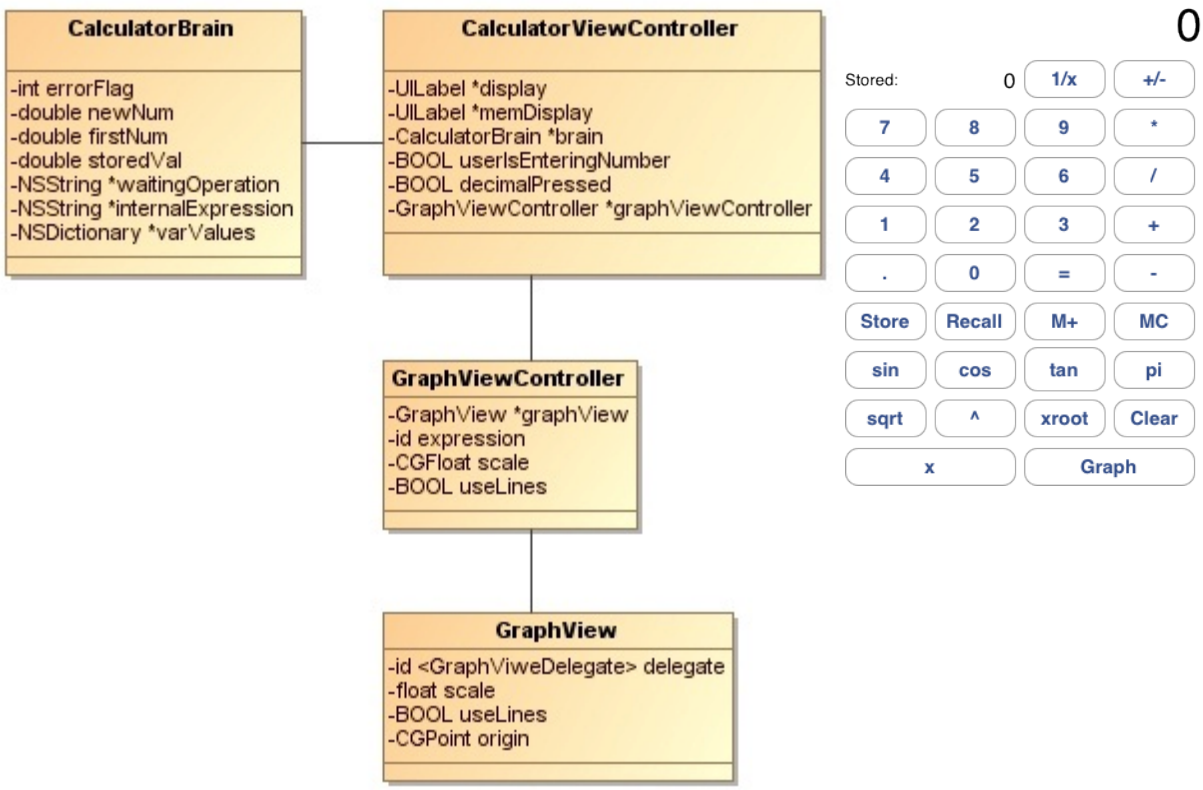
User Interface

Here is a look at the user interface for the calculator portion of the app, zeroed out:

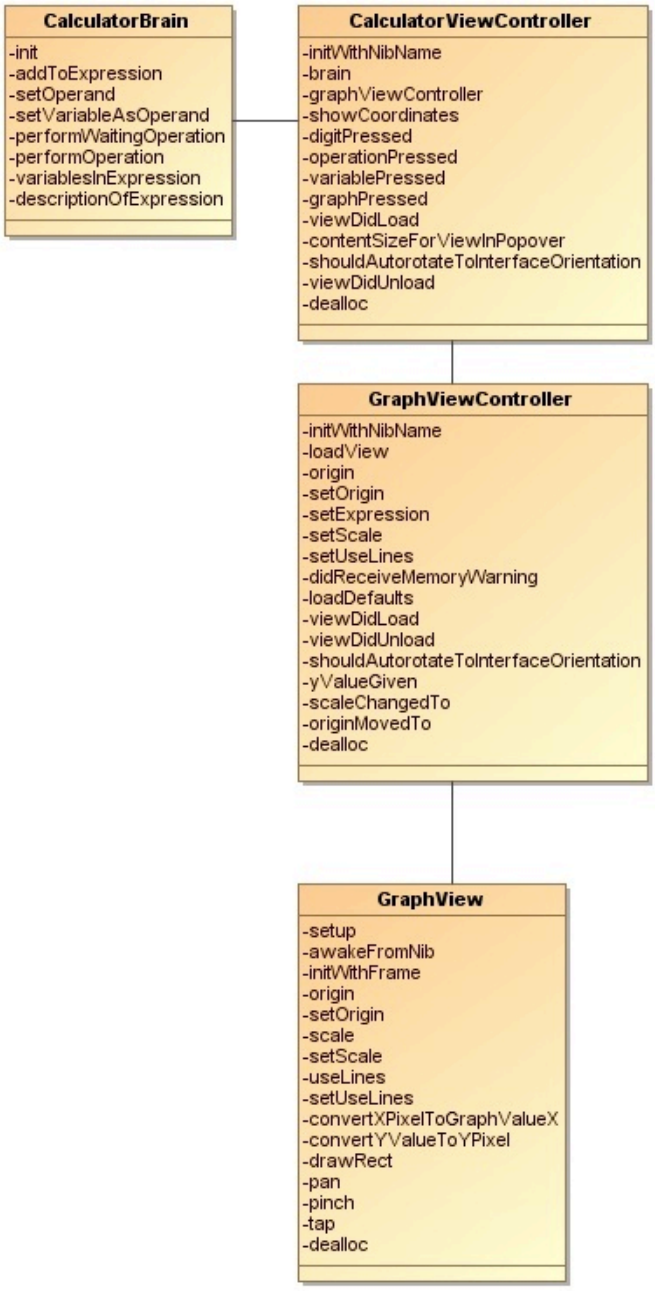


UML

Here is a look at the arrangement of the main classes involved in the calculator, and their variables:



Here is a similarly constructed view of the methods for the



classes.

The lines connecting these classes illustrates the MVC structure of the program, and they indicate where the lines of communication are open (and, Professor Downing, I need to add a line running from the CalculatorViewController class to the little image of the view.) The CalculatorViewController communicates with the calculator view, and the calculator brain, which is the model. It also communicates with the GraphViewController, which in turn communicates with the GraphView. The rest of the classes are isolated, thus creating a very organized structure in which the location of possible bugs is limited and predictable.

For the early calculator, some methods were provided, and once the graph was involved, the course webpage provides a class to draw the axes in the view. This class is used by the graph view.

Flickr App

Problem Description and Stages

As for the app that accesses content from Flickr, the structure is a bit different. The first assignment for this app asks students (after obtaining a Flickr API key) to build an app that accesses the top rated spots from which photos were recently taken and uploaded and then view the photos they're interested in. These assignments are largely about learning how to handle raw data from the web, and, for example, display the image contained in that raw data, as well as saving data more permanently for use the next time the user opens the app. Another purpose to these assignments is to learn to use the navigation based view controller, navigationController. This controller generates a tabbed viewing experience for the user, displaying multiple tab buttons along the bottom of the screen and allowing the user to tap to switch between them.

At first, this assignment makes use of the class NSUserdefaults, a class that can be used to store persistent data to restore certain information about presets the user prefers inside your app. Here, however, user data includes a list of the links the user has previously clicked inside the Flickr app. This is one more thing that should prove very useful to understand in iOS programming in general.The second assignment for this app adds a favorites page, and allows the user to fill it with their favorite photos that they have found through the app. In the process of implementing these features, it is necessary to stop using NSUserdefaults for this data, and to switch to using the Core Data persistent storage.

Components

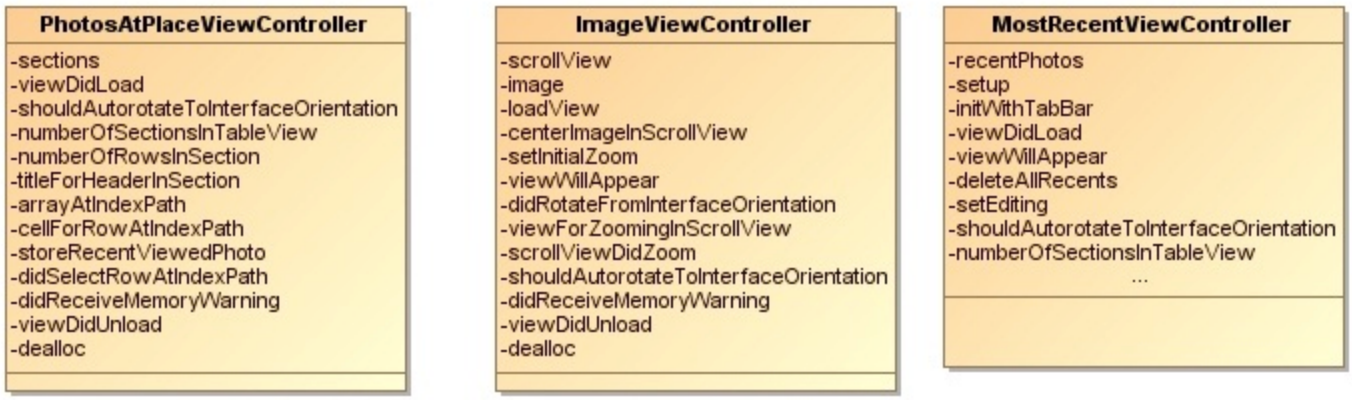
The main components for this assignment were separate view controllers that are all passed to the navigation controller. This includes: PhotosAtPlaceTableViewController, which manages the tab that displays a list of locations from which people have recently uploaded pictures; the MostRecentViewController, which displays a list of the images most recently viewed by the user; PhotoViewController and ImageViewController, which are used for actually displaying the photos the user chooses to view. This app looks less like it follows the standard MVC model, but it is more a demonstration of the "controllers of controllers" concept (which is part of the MVC model), since pretty much all these view controllers are controlled in turn by the navigation controller.

User Interface

UML

Here are the views of the class structure of the Flickr app:

And here is the set with the list of functions:



Provided Classes

One other class was provided, which was basically a set of methods to access Flickr data, called FlickrFetcher. This class includes methods for retrieving the lists of photos from Flickr, and for retrieving the photos themselves. The photos are sent as raw data, which I saved in the form of an instance of the built-in NSData class. It is necessary to call the method imageWithData in order to get the phone to interpret the raw data as an image.

Testing

Since the user does not directly enter any data, and instead simply clicks through to images and sets some for saving, testing was not very difficult for this set of assignments. The only place where there were issues was with extracting the data properly from the various pieces of information returned from the FlickrFetcher class. One interesting workaround that was necessary during testing was that Flickr seems to limit the number of queries in a certain timeframe. This meant that it was sometimes necessary to cache the results from the Flickr queries I would make in NSUserDefaults, in order to prevent the situation where Flickr simply returns zero results.

The Future

Now that I have been through this whole course, I feel pretty well equipped to approach the next task I have set out for myself. I do, however, have a great deal more reading to do concerning hearing aid algorithms and the audio libraries provided through iOS, and how to use them.

I have read that common hearing aid algorithms usually scale down higher

frequencies, since these are the ranges that people commonly lose first. This shifting provides the ear with enough information that the brain can piece together and make sense of the speech being heard. The first goal for this hearing aid app will be to implement such an algorithm for use in helping people to hear when in conversation over the phone. I may also add an option to use the effect on any sound produced by the phone's speakers, in order to improve peoples' ability to hear speech in any kind of media that includes audio.

After producing this basic application, which is not much more than a 'minimum viable product' demonstration, I will attempt to implement more features, including turning the app into a general-use hearing aid. The most advanced professionally produced hearing aids use multiple microphones in order to filter out noise and help people pick out what it is that they want to hear. Many of them accomplish this by amplifying the sound in front of a person, and noise-cancelling sound coming from behind. This requires directional microphones, but with the iPhone 4 (and I assume later models), this may be possible, because I have read that the iPhone 4 has a microphone on each end. If this is possible, it changes my hearing aid app from a piece of software that has uses in the context of using a phone, to an application that could help many people in their regular interaction with people in their daily lives.