Project 2
Evil Hangman*

Each milestone’s deadline is noon.
See cs164.net/expectations for each milestone’s expectations.

<table>
<thead>
<tr>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3/23 Proposal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3/27 Design Doc, Style Guide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4/3 Alpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4/13 Release</td>
</tr>
</tbody>
</table>

Help

Help is available throughout the week at http://help.cs164.net/, and we’ll do our best to respond within 24 hours. But do turn first to your partner with bugs!

* Inspired by Keith Schwarz of Stanford.
Academic Honesty

All work that you do toward fulfillment of this course’s expectations must be the work of you and your partner. Collaboration with anyone other than the partner with whom you begin the semester is not permitted unless one of the course’s heads approves a change of partner in writing. Partners must contribute equitably to each milestone: you may not implement most or all of some project’s milestone and submit it on behalf of your two-person team.

Viewing or copying another individual’s work (even if left by a printer, stored in an executable directory, or otherwise exposed) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to projects to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (e.g., for a job) must be approved by the staff.

You may read and comment upon classmates’ code toward fulfillment of projects’ code reviews but only for classmates whose code is assigned to you by the course’s staff for review. You may integrate ideas and techniques that you glean from your reviews of classmates’ code and from classmates’ reviews of your code into your own work, so long as you attribute those ideas and techniques back to your classmates, as with comments in your own code. As for classmates beyond your own partner and those with whom you’re involved in reviews, you may discuss projects, including designs, but you may not share code. In other words, you may communicate with those classmates in English, but you may not communicate in PHP, JavaScript, or Objective-C. If in doubt as to the appropriateness of some discussion, contact the course’s heads.

You may turn to the Web for instruction beyond the course’s lectures and labs, for references, and for solutions to technical difficulties, but not for outright solutions to projects or portions thereof. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course’s lectures and labs (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is Admonish, Probation, Requirement to Withdraw, or Recommendation to Dismiss, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.
Hangman.

☐ Well hello! I am thinking of a seven-letter word:

        __________

Guess a letter, and I’ll tell you where, if anywhere, it appears in the word. You can guess incorrectly up to, oh, six times. If you can figure out the word, you win; if you can’t (and you’re out of guesses), I win. Ready? Okay, guess a letter.

A, you say? You are so smart. That letter appears twice:

        -A---A-

Recognize the word now? No? Not to worry, you still have six chances left, since your first guess was a good one. Guess again.

E, you say? Nope, sorry, not so smart after all. You’ve now used up one of your six chances, so you have five chances left. Guess again.

I, you say? Nope! You have four chances left. Guess again.

Z, you say? Nope! You now have three chances left. Guess again.

N, you say? Nice! That letter also appears twice:

        -AN--AN

You still have three chances left. Guess again.

E, you say? I’ll pretend I didn’t hear that. You guessed that already! You still have three chances left. Guess again.

H, you say? Nice! That letter appears once:

        HAN--AN

Recognize the word yet? G, you say? M, you say? That’s right! It’s HANGMAN!

☐ So that is how the age-old game of Hangman is played. But for this project, you will not just implement Hangman. (Hangman is boring.) You will implement….

Wait for it…

Evil Hangman.
Evil Hangman.

Evil Hangman is quite like Hangman, except that the computer cheats. Rather than pick, say, a seven-letter word at the game’s start, the computer instead starts off with a mental list (well, maybe a set or array) of all possible seven-letter English words. Each time the human guesses a letter, the computer checks whether there are more words in its list with the letter than without and then whittles the list down to the largest such subset. If there were more words with the letter than without, the computer congratulates the user and reveals the letter’s location(s). If there were more words without the letter than with, the computer just laughs.

Put more simply, the computer constantly tries to dodge the user’s guesses by changing the word it’s (supposedly) thinking of. Pretty evil, huh? The funny thing is most humans wouldn’t catch on to this scheme. They’d instead conclude they’re pretty bad at Hangman. Mwah hah hah.

Suffice it to say that the challenge ahead if you is to implement...

Wait for it...

Evil Hangman.

But what is the algorithm for evil? Well, let’s consider what the computer needs to do anytime the user guesses a letter. Suppose that the user needs to guess a four-letter word and that there are only a few such words in the whole English language: BEAR, BOAR, DEER, DUCK, and HARE. And so the computer starts off with a universe (i.e., list) of five words. Next suppose that the user guesses E. A few of those words contain E, so the computer had best decide how to dodge this guess best. Let’s partition those words into “equivalence classes” (a la CS121) based on whether and where they contain E. It turns out there are four such classes in this case:

----- , which contains BOAR and DUCK
- -E-- , which contains BEAR
- -EE-- , which contains DEER
- - - - -E , which contains HARE

Note that HARE is not in the same equivalence class as BEAR. Even though both have just one E, each’s E is in a different location. If the computer is forced to admit that the word that it’s “thinking of” contains an E (because it just so happens to have discarded all words that lack E in response to the user’s past guesses), it will be forced to announce (and thus commit to) that letter’s location.

So, back to our story: the user has guessed E. What should the computer now do? Well, it could certainly declare that the word it’s “thinking of” does not contain E, the implication of which is that the list of five words becomes two (BOAR and DUCK). That does feel optimal: armed with two words, the computer might still have a chance to “change its mind” yet again later. Then again, DUCK seems pretty easy to guess, whereas most people might not even think of HARE. On the other hand, I don’t remember the last time I used BOAR in a sentence. But let’s keep it simple: you
may assume that the computer will always react to a user’s guess by whittling its list down to the largest equivalence class, with pseudorandomness breaking any ties.

Realize, though, that the largest equivalence class might not always correspond to -----. For instance, suppose that the user had guessed R instead of E. The equivalence classes would thus be as follows:

-----, which contains DUCK
----R, which contains BEAR, BOAR, and DEER
---R-, which contains HARE

In this case, the computer should go ahead and admit that the word that it’s “thinking of” contains R at its end, since that leaves three possible words (BEAR, BOAR, and DEER) and thus the maximum amount of maneuverability down the road in reaction to subsequent guesses. Of course, if the user has never heard of a DUCK, then ---- could very well be a superior strategy. But, again, lest you drive yourself nuts with overanalysis, you may assume that the largest equivalence class is optimal, even though it might not be in reality.

Indeed, that strategy might sometimes backfire, at least in a sense. Suppose that, in a new version of the story at hand, there are only three four-letter words in the English language: BOAR, DEER, and HARE. Suppose now that the user guesses E. In this case, our equivalence classes are:

-----, which contains BOAR
--EE-, which contains DEER
---E, which contains HARE

Because each has one word, these classes are, of course, of the same size. But if we happen to select --EE-- pseudorandomly, thereby whittling our list down to just DEER, we’ll have revealed to the user two of the word’s letters, whereas we could have revealed zero (had we selected ---- instead) or one (had we selected ----E instead). But, again, that’s okay. You are welcome, but not required, to implement a more sophisticated algorithm than that prescribed here; just make clear in some comments how yours happens to work.

As for how to divide a list of words into equivalence classes programmatically, well, we’ll leave that as a challenge for you!

System Requirements.

☐ From this point forward in the course, you’ll need (access to) an Intel-based Mac running Lion (Mac OS X 10.7.3 or later). To find out whether a particular Mac qualifies, select About This Mac from the  menu. Hopefully you’ll see Version 10.7.3 (or later) as well as some mention of Intel next to Processor.
If you don’t own a Mac that meets these requirements, see https://www.cs164.net/FAQs for answers to some FAQs. In particular, note that you’re welcome to use the Macs in Science Center 418d, per the schedule at http://bioinformatics.fas.harvard.edu/bioinformatics/policies.html.

Required Reading.

☐ First curl up with Learning Objective-C: A Primer:

You’ll find that it’s a pretty quick read and hopefully whets your appetite for a bit more detail.

☐ It’s time for more detail! Now curl up with The Objective-C Programming Language:

This one’s a few chapters, so be sure to click through to each by clicking Next in the bottom-right corner of most every page or by clicking through to each via the Table of Contents in the site’s top-left corner.

Odds are you won’t retain everything you read. But not to worry; it’ll start to sink in once you get your hands dirty with code of your own.

☐ Next skim Coding Guidelines for Cocoa:

Try to keep those guidelines in mind as you begin to write code of your own. We won’t expect strict adherence to Apple’s guidelines, so long as your own style is clean and consistent, but might as well familiarize yourself with some best practices.

Register as an Apple Developer.

☐ Phew, that was a lot of reading. Time to fill out some forms! Head to

if you’re not already registered as an Apple Developer. Click Get Started when ready, and you should be prompted to Create an Apple ID or Use an existing Apple ID. Review the explanation beneath each option, select the appropriate one, click Continue, then follow the on-screen prompts.

If you plan to submit an app to Apple’s App Store, whether during the semester or shortly thereafter, you might also want to sign up for the iOS Developer Program at
which costs $99/year or for the iOS Developer Enterprise Program at

which costs $299/year. However, know that you do not need to sign up for either program for this course. Because the course is part of the iOS Developer University Program, you will be able to install apps that you write on your own iPad, iPhone, or iPod touch this semester for free. You won’t be able to submit any apps to the App Store, though, unless you sign up for one of the paid programs. See http://developer.apple.com/programs/ which-program/ for more details.

Xcode.

If, at this very moment, using a Mac in Science Center 418d, where everything’s already installed for you, go ahead and skip the following two checkboxes. Otherwise it’s time to install Xcode!

Assuming you only signed up as an Apple Developer (for free) and neither of the paid programs, visit

with Safari, even if not your preferred browser, then click View in Mac App Store at top-left. That button should trigger the Mac’s own App Store to launch, at which point you can download Xcode 4.3.1 for free. Do so and proceed to install it once downloaded.†

If you did sign up for the iOS Developer Program (for $99/year) or the iOS Developer Enterprise Program (for $299/year), head to
http://developer.apple.com/xcode/

and click Log in where prompted in order to download Xcode 4.3.1 for free. (Well, “free,” seeing as you did just pay $99 or $299.) Proceed to install it once downloaded.’

No matter how you download Xcode, realize that the installer is over 4GB in size, so it might take some time!

Once Xcode is installed, go ahead and launch it. (It can be found on your Mac’s startup volume in /Applications/.)† Proceed to select Preferences... under the Xcode menu (to the right of the ▼ menu). Click the Downloads icon atop the window that appears, then select Components below it. Highlight Command Line Tools, then click Install, providing your Mac’s password if prompted.

† If you already have an earlier version of Xcode installed, Xcode 4.3.1’s installer will allow you to retain or remove it.
‡ Prior versions of Xcode could be found in /Developer/Applications/.
Hello, World.

☐ Alright, it’s time to take Xcode for a spin. Go ahead and launch it, if not running already, and create a new project, as by clicking Create a new Xcode project in the splash screen that displays upon launch (if you didn’t disable) or by selecting New > New Project... from Xcode’s File menu. When prompted to choose a template for your new project, select Application under Mac OS X (not iOS), select Command Line Tool, then click Next. On the screen that appears: input MacApp for Product Name; input edu.harvard.college for Company Identifier; select Foundation (not Core Foundation) next to Type; then click Next. Choose a location for the project when prompted, check the box to create a local git repository, then click Create.

A window entitled MacApp.xcodeproj should then appear. Go ahead and expand each of the triangles at left (except for Foundation.framework), and you should see main.m, MacApp.1, and MacApp-Prefix.pch among their contents. Click MacApp.1 then hit delete on your keyboard; when prompted to permanently delete it, click Delete. (That file’s just a template for a “man page” that you won’t need for this project. See http://en.wikipedia.org/wiki/Man_page if unfamiliar.)

Now click main.m, and you should see its contents at right. Ha, turns out Xcode already wrote this program for you! But go ahead and change

@"Hello, World!"

to some other NSString of your choice. Then modify the comments atop the file so that they contain your name and your partner’s name. Henceforth, be sure that main.m always contains at least those details for any project you write.

Go ahead and click Run in Xcode’s top-left corner (or hit ⌘-R on your keyboard), and you should find that Xcode’s Debug area appears at bottom, among whose boldfaced output should be your NSString! You’ve just compiled and run MacApp.app, your very first app!

There’s nothing wrong with leaving that Debug area open all of the time, but, so that you get a feel for Xcode’s UI, go ahead and figure out how to hide it. (Hint: poke around Xcode’s top-right corner.) In fact, poke around the rest of Xcode’s UI, including its menu, to get a sense of its features. (If you start to feel a bit overwhelmed, simply close whatever you opened!) The interface is a bit complex, at least when you have everything showing, but not to worry, we’ll point out its features as needed.

☐ Okay, that first app isn’t going to impress any of your friends. Let’s try a bit harder.

Create a new project in Xcode. (Remember how?) When prompted to choose a template for your new project, select Application under iOS (not Mac OS X), select Single View Application, then click Next. (A Single View Application is among the simplest of the templates provided.) On the screen that appears: input iPhoneApp for Product Name; input edu.harvard.college for Company Identifier; leave Class Prefix blank (it’s okay if you see a placeholder of XYZ in gray); select iPhone next to Device Family; leave Use Storyboards unchecked; check Use Automatic Reference
Counting: leave Include Unit Tests unchecked; then click Next. Choose a location for the project when prompted, check the box to create a local git repository, then click Create.

A window entitled iPhoneApp.xcodeproj should then appear. As before, go ahead and expand each of the triangles at left (except for UIKit.framework, Foundation.framework, and CoreGraphics.framework), and you should see more files than last time.

Go ahead and click main.m, and its contents should appear to the right. As before, it’s that file that will kick off this program. Notice that it calls UIApplicationMain. Hold down option on your keyboard, click the name of that function, and some documentation thereof should appear in a callout. (If you’d like even more detail, click the little book icon in that callout’s top-right corner, and the entire UIKit Framework’s documentation will appear inside of Organizer.) Apparently, UIApplicationMain creates “the application object and the application delegate,” the latter of which is simply an object to which control of your application is “delegated.” How does it know what kind of object to create for delegation? Via its fourth argument! Indeed, notice how the fourth argument to UIApplicationMain in main.m mentions AppDelegate, which just so happens to be a class declared in AppDelegate.h and defined in AppDelegate.m (both of which are provided for you via the Single View Application template).

Now take a peek at AppDelegate.m. Notice how it allocates a “view controller” in application:didFinishLaunchingWithOptions: and then initializes it with a nib (i.e., ViewController.xib). Interesting!

Go ahead and click ViewController.xib, and Xcode’s “interface builder” should appear to the right. Hmm, not that interesting after all; it’s just a blank view. Let’s change that!

In Xcode’s top-right corner, click the rightmost icon above View to show Xcode’s Utilities if not visible already. Toward that area’s bottom should be a library of Objects (assuming you have the little cube icon selected), the first of which is Label. Drag a Label from that area to the center of the view atop the graph paper in Xcode’s middle (i.e., the iPhone-sized rectangle). Crosshairs should appear once you’ve aligned the Label perfectly. Assuming the Label is selected (as indicated by a rectangle of six squares around it), select the Attributes inspector among Xcode’s Utilities at right. (Hover over each of the icons below View in Xcode’s top-right corner to find that inspector.) Then specify some Text, a Font, and a Text Color (as well as any other attributes that you’d like) for your Label. Any changes you make should appear in the graph paper’s window.

Once pleased with your Label, ensure that iPhoneApp > iPhone 5.1 Simulator is selected in the drop-down to the right of the Run button in Xcode’s top-left corner. Then click Run (or hit ⌘-R on your keyboard). If all goes well, the iOS Simulator should launch with your app! If not, try to retrace your steps to determine what might have gone wrong. (Worst case, perhaps start over from the beginning!)

What about the other files that come with this template? Well, ViewController.h and ViewController.m (along with ViewController.xib) collectively govern the behavior of this application’s view controller. Since our only changes to the template were aesthetic, it sufficed to make them in the nib. In iPhoneApp-Info.plist and InfoPlist.strings are some
(default) properties for your app. In iPhoneApp-Prefix.pch are some headers that will be prepended to each of your source files. As for iPhoneApp.app under Products, that's your app. Or, at least, it will be if you build your code for an actual device (which you don’t need to yet); for now, it’s probably red, which means you’ve only built your app for the iOS Simulator.

Phew, nicely done!

☐ Okay, realistically, neither of those apps are going to impress your friends. You can delete both if you’d like. (You won’t need to submit them.) Was a good warm-up, though!

Bitbucket.

☐ Okay, for the sake of discussion, we again need to call you or your partner Alice and the other of you Bob. Assume the same identities that you assumed on for Project 1, then read on!

☐ Only Alice should perform this step.

Log into your Bitbucket account and create a new, private repo as follows:

☐ Select Repositories > create repository at top-right.
☐ Ensure that Create new repository is highlighted (in dark blue).
☐ Input a value of project2 under Name.
☐ Ensure that Private is checked.
☐ Ensure that Git is selected under Repository type.
☐ Check both Issue tracking and Wiki under Project management.
☐ Select Objective-C under Language.
☐ Input a value for Description and/or Website if you’d like.
☐ Click Create repository.

You should then find yourself at a page whose URL is https://bitbucket.org/alice/project2, where alice is your actual Bitbucket username. Click the Admin tab at top, then click Access management at left. In the text field under Users (1), input your partner’s Bitbucket username, then click Admin at right. Next, input cs164 into that same text field, then click Admin at right. Finally, input your TF’s username (which can be found at https://www.cs164.net/Staff), then click Admin at right. Your partner and CS164’s staff, including your TF, should now have access to your repository. Your partner can confirm as much by visiting https://bitbucket.org/alice/project2, where alice is your actual Bitbucket username.
Only Alice should perform this step too.

Launch Xcode and select File > New > Project.... When prompted to choose a template, select Application under iOS (if not selected already), then select Utility Application. Then click Next.

On the screen that appears: input project2 for Product Name; input edu.harvard.college for Company Identifier; leave Class Prefix blank (it’s okay if you see a placeholder of XYZ in gray); select iPhone next to Device Family; leave Use Storyboards unchecked (unless you’d like to use them); leave Use Core Data unchecked (unless you’d like to use it); check Use Automatic Reference Counting; check Include Unit Tests; then click Next. Choose a location for the project when prompted, check the box to create a local git repository, then click Create.

Select File > Source Control > Repositories..., and Organizer should open. Select Remotes under project2 in Organizer’s lefthand menu, then click Add Remote toward the window’s bottom. On the screen that appears, input origin for Remote Name, and input https://alice@bitbucket.org/alice/project2.git for Location, where alice is your actual Bitbucket username, then click Create. A folder called origin should then appear in Organizer. With origin selected, input your Bitbucket password toward the window’s bottom. Then close Organizer’s window.

Return to Xcode itself and, with project2.xcdeproj in the foreground again, select File > Source Code > Push.... To the right of Remote should be origin/master, and a green light should indicate that the repository is online. (If not, best to retry all these steps.) Click Push. If informed that your push was successful, visit

https://bitbucket.org/alice/project2/src

with a browser, where alice is your actual Bitbucket username. You should see your code!

Henceforth, anytime you’d like to commit some change(s) to your local repository (on your own hard drive), select File > Source Control > Commit... and follow the prompts. Anytime you’d like to push some change(s) to Bitbucket (perhaps for your partner’s sake), select File > Source Control > Push... as you just did. And anytime you’d like to pull some change(s) from Bitbucket (as from your partner), select File > Source Control > Pull.... Alternatively, you can use a command line by opening Applications > Utilities > Terminal, navigating your way to your project2/ directory (as via cd), and using git as usual.⁶

---

⁶ Recall that you created an SSH key pair for Bitbucket inside of the CS50 Appliance for Project 0. If you’d like to use git at the command line, you’ll want to copy that key pair (e.g., ~/.ssh/id_rsa and ~/.ssh/id_rsa.pub) from the appliance to your Mac’s ~/.ssh/ directory, or you’ll want to create a new key pair on your Mac (as with ssh-keygen) and then add the newly created public key to your Bitbucket account.
Okay, now Bob should perform this step.

Launch Terminal, as via Applications > Utilities > Terminal. Navigate your way to wherever you’d like to store Project 2’s code (e.g., ~/Documents/), as via cd. Then execute

```
git clone https://bob@bitbucket.org/alice/project2.git
```

where bob is your actual Bitbucket username and alice is Alice’s Bitbucket username, inputting your Bitbucket password if prompted. (In theory, you can clone repositories in Xcode itself, but cloning appears to be buggy in Xcode 4.3.1, at least for HTTPS.)

Once done cloning, launch Xcode, as via Applications > Xcode. If prompted to choose a template for a new project, click Cancel. Then select File > Open..., and navigate your way to your project2 directory. Select project2.xcodeproj, then click Open. You should see the codebase that Alice pushed to Bitbucket!

Henceforth, anytime you’d like to commit some change(s) to your local repository (on your own hard drive), select File > Source Control > Commit... and follow the prompts. Anytime you’d like to push some change(s) to Bitbucket (perhaps for your partner’s sake), select File > Source Control > Push... as you just did. And anytime you’d like to pull some change(s) from Bitbucket (as from your partner), select File > Source Control > Pull.... Alternatively, you can use a command line by opening Applications > Utilities > Terminal, navigating your way to your project2 directory (as via cd), and using git as usual.”

** Tour. **

Shall we take a tour of the codebase, copies of which you and your partner both now have?

With project2.xcodeproj open in Xcode, go ahead and expand each of the triangles at left (except for UIKit.framework, Foundation.framework, CoreGraphics.framework, and SenTesting.framework), and you should see AppDelegate.{h,m}, MainViewController.{h,m,xib}, and FlipsideViewController.{h,m,xib} among their contents. We’ll take a tour through each of those files, but first a quick demo. Ensure that project2 > iPhone 5.1 Simulator is selected in the drop-down to the right of the Run button in Xcode’s top-left corner. Then click Run (or hit ⌘-R on your keyboard). If all goes well, the iOS Simulator should launch with this app. How fun!

Okay, it doesn’t do all that much yet, but do click that ① button in the app’s bottom-right corner. Notice how the app flips around to its flipside. Well that’s kind of neat. Click Done in the flipside’s top-left corner, and you should be returned to the front side. It’s okay if you’d like to do that again.

**Ibid.**
Anyhow, once ready for that tour of the code, quit iOS Simulator, return to Xcode, and open up AppDelegate.h. Notice how this file declares a class called AppDelegate, which implements a protocol called UIApplicationDelegate. That’s good, because it’s to an object of that class that main (in main.m) will ultimately delegate control. That class is implemented in AppDelegate.m, most of whose methods are just stubs with comments, except for applicationWillFinishLaunchingWithOptions:

Now open up MainViewController.xib. Ah, there’s part of the app’s UI. Indeed, there’s that ⌘ button. If you see two cubes and one square to the left of the graph paper in Xcode’s middle, click the little arrow in the circle in the graph paper’s bottom-left corner. Then click File’s Owner under Placeholders. Can you guess who the owner of this nib will be? Open Xcode’s Identity Inspector to confirm or deny your prediction, as by clicking the fourth icon from the right below View in Xcode’s top-right corner. Yup, this nib belongs to an instantiation of the MainViewController class.

Now click View under Objects (which itself is below Placeholders). Notice that this object’s class is UIView, and it looks like it will fill an iPhone’s whole screen. To confirm as much, open Xcode’s Attributes Inspector, as by clicking the third icon from the right below View in Xcode’s top-right corner, and change Background to something other than gray. (You may need to click the graph paper to see the change.)

Now open up MainViewController.h, which declares the class that owns the nib you just closed. Notice how MainViewController descends from UIViewController. That’s handy, because UIViewController comes with a whole lot of features, per the UIViewController Class Reference:


Notice, too, that MainViewController implements a protocol called FlipsideViewControllerDelegate. (It turns out that protocol is defined in FlipsideViewController.h, but more on that in a moment.) Notice that MainViewController has one instance method (showInfo:), but no instance variables.

Now open up MainViewController.m. Hm, lots more in this file. Let’s look at each method in turn. Defined first are viewDidLoad, viewDidUnload, and shouldAutorotateToInterfaceOrientation:, all of which are documented in that UIViewController Class Reference. Also defined in MainViewController.m is

FlipsideViewControllerDidFinish:, which belongs to that protocol called FlipsideViewController, but more on that in just another moment. Last up is showInfo:, whose declaration was in MainViewController.h. Notice how this method allocates a FlipsideViewController, thereafter initializing it with FlipsideViewController.xib. It then defines itself as the delegate for the flipside’s controller, sets the transition from front side to flipside, and presents that flipside.

Alright, almost done with the tour. Open up FlipsideViewController.xib. Ah, there’s the flipside’s UI. If you ctrl- or right-click File’s Owner, you’ll see how this nib’s owner (a FlipsideViewController) is wired up to a view. You’ll also see how that Done button is
wired to a method (i.e., UIAction) called done:. If you expand the triangle next to View (and, in turn, everything below), you’ll see how the flipside’s navigation bar and title relate to that button. In fact, if you ctrl- or right-click Bar Button Item — Done, you’ll see that same wiring in reverse.

Okay, now open up FlipsideViewController.h. Not only does this file declare a class called FlipsideViewController (which descends from UIViewController), it declares a property called delegate, which must apparently point to an object (implied by id) that implements the protocol called FlipsideViewControllerDelegate. It also declares that instance method called done:. And, as promised, it declares that protocol, which apparently prescribes a method called flipsideViewControllerDidFinish:. The implication, of course, is that any class that implements FlipsideViewControllerDelegate (e.g., MainViewController) should implement a method called flipsideViewControllerDidFinish:. Just as we’ve seen!

Finally, open up FlipsideViewController.m. In here are some stubs, just like those we saw in MainViewController.m. But this file also implements done:, whose sole line of code apparently informs the flipside’s delegate (i.e., a MainViewController object) when the flipside is done flipping back.

Phew, time for a break.

Welcome back. Feel free now to poke around the files in the Supporting Files group, but odds are you won’t need to touch any of them for this project. But let’s add one additional file to that group, an array of 234,371 English words! Download the property list (i.e., XML file) at

http://cdn.cs164.net/2012/spring/projects/2/words.plist?download

and then drag words.plist into that Supporting Files group within Xcode. When prompted to choose options for adding the file, check Copy items into destination group’s folder (if needed), ensure that project2 is checked to the right of Add to targets, and then click Finish. (Since words.plist isn’t a folder, it doesn’t matter what’s selected next to Folders in that window.) You should now see words.plist among your app’s Supporting Files. Click it, and you’ll see a huge array of English words.

Alright, that’s it! It’s time to delegate control of this project to you! (Get it?)

Specification.

Your challenge for this project is to implement Evil Hangman as a native iOS app. The overall design and aesthetics of this app are ultimately up to you, but we require that your app meet some requirements. All other details are left to your own creativity and interpretation.

†† The words are taken from /usr/share/dict/words on Mac OS 10.7.3, with all forced to uppercase and duplicates removed.

‡‡ Even though Xcode presents the contents of words.plist with three columns (Key, Type, and Value) as though the file contained a dictionary, it indeed contains an array (implemented in XML with an array element, which you can see if you open words.plist with a text editor).
Features.

☐ Immediately upon launch, gameplay must start (unless the app was simply backgrounded, in which case gameplay, if in progress prior to backgrounding, should resume).
☐ Your app’s front side must display placeholders (e.g., hyphens) for yet-unguessed letters that make clear the word’s length.
☐ Your app’s front side must inform the user (either numerically or graphically) how many incorrect guesses he or she can still make before losing.
☐ Your app’s front side must somehow indicate to the user which letters he or she has (or, if you prefer, hasn’t) guessed yet.
☐ The user must be able to input guesses via an on-screen keyboard.
☐ Your app must only accept as valid input single alphabetical characters (case-insensitively). Invalid input (e.g., multiple characters, no characters, characters already inputted, punctuation, etc.) should be ignored (silently or with some sort of alert) but not penalized.
☐ Your app’s front side must have a title (e.g., Hangman) or logo as well as two buttons: one that flips the UI around to the app’s flipside, the other of which starts a new game.
☐ If the user guesses every letter in some word before running out of chances, he or she should be somehow congratulated, and gameplay should end (i.e., the game should ignore any subsequent keyboard input). If the user fails to guess every letter in some word before running out of chances, he or she should be somehow consoled, and gameplay should end. The front side’s two buttons should continue to operate.
☐ On your app’s flipside, a user must be able to configure three settings: the length of words to be guessed (the allowed range for which must be [1, n], where n is the length of the longest word in words.plist); the maximum number of incorrect guesses allowed (the allowed range for which must be [1, 26]); and whether or not to be evil. By default, your app must be evil. But if the user opts to disable evil, gameplay should occur in a traditional, non-evil way, whereby the app must choose a word pseudorandomly from the start and stay committed to that word until the game’s end.
☐ When settings are changed, they should only take effect for new games, not one already in progress, if any.
☐ Your app must maintain a history of high scores that’s displayed anytime a game is won or lost. We leave the definition of “high scores” to you, but you should somehow rank the results of at least 10 games (assuming at least 10 games have been won), displaying for each the word guessed and the number of mistakes made (which is presumably low). The history of high scores should persist even when your app is backgrounded or force-quit.

Implementation Details.

☐ Your app’s UI should be sized for an iPhone or iPod touch (i.e., 320 × 480 points) with support for, at least, UIInterfaceOrientationPortrait. However, if you or your partner owns an iPad and would prefer to optimize your app for it (i.e., 768 × 1024 points), you may, so long as you inform your TF prior to this project’s deadline.
☐ You must use the contents of words.plist as your universe of possible words. You’re welcome, but not required, to transform it into some other format (e.g., SQLite).
Your app must come with default values for the flipside’s two settings; those defaults should be set in NSUserDefaults with registerDefaults:. Anytime the user changes those settings, the new values should be stored immediately in NSUserDefaults (so that changes are not lost if the application is terminated).

You must implement each of the flipside’s numeric settings with a UISlider. Each slider should be accompanied by at least one UILabel that reports its current value (as an integer).

You must implement the flipside’s evil toggle with a UISwitch.

You are welcome to implement your UI with Xcode’s interface builder in MainViewController.xib and FlipsideViewController.xib, or you may implement your UI in code in MainViewController.{h,m} and FlipsideViewController.{h,m}.

You must obtain a user’s guesses via a UITextField (and the on-screen keyboard that accompanies it). For the sake of aesthetics, you are welcome, but not required, to keep that UITextField hidden (so long as the on-screen keyboard works). You are also welcome, but not required, to respond to user’s keypresses instantly, without waiting for them to hit return or the like, in which case textField:shouldChangeCharactersInRange:replacementString in the UITextFieldDelegate protocol might be of some interest.

You must implement your app’s two strategies for gameplay (evil and non-evil) in two separate model classes called EvilGameplay and GoodGameplay (in files called EvilGameplay.{h,m} and GoodGameplay.{h,m}, respectively) both of which must implement a protocol called GameplayDelegate (which must be declared in a file called GameplayDelegate.h). In other words, based on whether evil is enabled or disabled, your app should pass messages to an instance of one class or the other.

You must implement the display of high scores in a UIViewController called HistoryViewController (in files called HistoryViewController.{h,m,xib}) that presents itself at game’s end via a UIModalTransitionStyleCoverVertical transition. You must also declare a HistoryViewControllerDelegate protocol (in HistoryViewController.h) that MainViewController implements, much like FlipsideViewController.h declares FlipsideViewControllerDelegate.

You must implement methods with which to store and retrieve high scores in a model called History (as by creating History.{h,m} files). You must store high scores persistently, as in a property list (other than words.plist) or in some other format (e.g., SQLite).

Your app must use Automatic Reference Counting (ARC).

You must implement unit tests for your models.

Your app must work within the iPhone 5.1 Simulator; you need not test it on actual hardware. However, if you or your partner owns an iPad, iPhone, or iPod touch, and you’d like to install your app on it, see https://manual.cs50.net/iOS for instructions.