**CSCI E65g: Mobile Application Development Using Swift and iOS**

**Fall 2018**

**Assignment 7**

**Issued: 10/17/2018 Due: 10/30/2018 NOTE TWO WEEKS**

**Scoring**

Graduate: 350 points

**Repository**

Find through [Github Invitation](https://classroom.github.com/a/APsKp5EF).

**Overview**

You have laid the foundation for the final project. It will now be made increasingly realistic (and as a result, useful, enjoyable and shareable), as far as we can get in the remaining 6 weeks.

The first phase we will add two key components: **Customizability** and **Persistence**. The persistence aspect will allow us to save and restore the game even when the App quits and restarts (which can happen for many reasons). The customizability aspect will allow the user to change the appearance to suit their individual taste and identity. In particular we will be able to choose colors and player names using a preferences dialog.

Persistence will make us learn the FileManager and UserDefaults APIs, both methods of different complexity levels for storing data on the device even when it is completely powered down.

Customizability will make us learn multi-screen Apps, involving transitioning from one View Controller to another and back. Key concepts are the View Controller lifecycle, Segues, and layered MVC (probably the trickiest aspect).

We will try to get practice with 2 of the 3 main ways of making a multi-screen App:

1. Manual segue
2. Navigation Controller segues
3. Tabbed View Controller (no segues)

**Reading**

**Please note:** not all of the code referenced below as of 2018/10/17 has been published yet. Please be patient and start with reading and the non-persistence aspects first.

* Segues
  + [Apple Introduction](https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html) (you can ignore shouldPerformSegue, unwind segues, and creating custom segues)
  + [Detailed classification of segues](https://digitalleaves.com/segues-navigation-ios-basics/) although focus on the one we’re sticking with, the **Push** segue.
* UINavigationController
  + [Apple Introduction and Reference](https://developer.apple.com/documentation/uikit/uinavigationcontroller)
  + [Basic tutorial similar to in-class](https://www.simplifiedios.net/ios-uinavigationcontroller-tutorial/) except these are embedded in a Navigation Controller
* UITabViewController These are fairly simple and don’t need much more than [the reference](https://developer.apple.com/documentation/uikit/uitabbarcontroller).
* UserDefaults The mechanism for saving basic preference data and very simple data models to disk with minimal effort.
  + [Apple reference](https://developer.apple.com/documentation/foundation/userdefaults)
  + [Hacking with Swift Tutorial](https://www.hackingwithswift.com/read/12/2/reading-and-writing-basics-userdefaults)
  + A [somewhat more advanced](https://medium.com/swift-programming/swift-userdefaults-protocol-4cae08abbf92) discussion on maintaining good coding style when using UserDefaults. Stop at **API Goals** where it gets too obscure.
* FileManager for storing arbitrary files on disk. *We only care about this API for determining a file path*, which is passed on to the archiving API, below.
  + [Apple discussion and reference](https://developer.apple.com/documentation/foundation/filemanager)
  + [Simple introductory reference](https://www.techotopia.com/index.php/Working_with_Files_in_Swift_on_iOS_8) and [runnable App](https://www.techotopia.com/index.php/IOS_8_Directory_Handling_and_File_I/O_in_Swift_%E2%80%93_A_Worked_Example)
* NSKeyedArchiver (and the [NSCoding](https://developer.apple.com/documentation/foundation/nscoding) protocol) for turning Swift objects into a stream of data to store on disk, and restoring them.
  + [Apple Introduction and reference](https://developer.apple.com/documentation/foundation/nskeyedarchiver)
  + [Pretty good tutorial on using it](https://medium.com/yay-its-erica/using-nskeyedarchiver-to-persist-data-976ab2f28006)
  + [More advanced discussion](https://nshipster.com/nscoding/) Mostly useful for the code listing where a more complex object is made compliant with the archiver. (Note subsequent listings show a way to store archive data in UserDefaults rather than the file system, which we will not be doing.)

**Problems**

1. (30 pts)

We will be supporting user preferences to customize the appearance and behavior of the App. First let’s worry about the underlying data representation.

In the previous assignment you were supplying colors from the View Controller directly. We didn’t really care if these were hardcoded in a switch statement or some other table of constants. But now it’s time to lift these out of being trapped in code, and add some more personalization besides. Players would like to use their own names.

We need to grow a level of sophistication and realize the data model needs to be represented by more than one class. Your new GamePrefsModel will store this changeable, important information, but it’s totally independent of central game logic, like grid size and rules — so it needs to be in its own class. (Astute readers will note there is a dependency: the game determines how many players are allowed, and the preferences model must support that number. However, without losing much generality, we’re fixing the entire project at **two** players through the semester.) It’s not a big deal for the View Controller to track two model references, each with a distinct purpose. If we had five or six we might need to wrap them in a consolidation layer but we max out at two model objects for the semester as well.

Create the GamePrefsModel class and the corresponding protocol. Update the View Controller to draw the player names and colors from it, as always through its protocol published capabilities only.

1. (20 pts)

Create the GamePrefsListener protocol just like you created a listening protocol for the Game Model in Assignment 6. The View Controller must subscribe and update itself whenever the model has changed. Consider two separate methods: namesChanged: and colorsChanged: so the updates can be focused on one or the other. Consider using property observers (didSet) in the model implementation as an easy way to monitor changes. Test it by hard-coding an arbitrary model change (to colors AND names) on an arbitrary event: viewDidAppear. This way you get a little life cycle practice as well. No short-circuiting though: test the full MVC propagation by calling a published method on the model only.

1. (50 pts)

Where there’s a model, there’s a UI. You should have gotten that sinking feeling that those new properties will need a UI so the players can edit them.

Create a simple preferences screen where each of the four values can be changed. The names are simple: the natural UI is a UITextField. For the colors, use aUIButton programmatically filled with the background color currently set in the model. Make it about the same size as the name field. Label everything clearly. The preferences controller will need a stored preferences model reference, naturally. For now, just hard-code an arbitrary one that proves the UI is following the model. Now this controller can be tested in isolation (unit test!).

1. (45 pts)

Make the color editable connecting a Storyboard **show** segue from each player color button to the ColorPickerVC class. (This class will be supplied for you as starter code. It is not part of iOS so we had to roll our own.) The ColorPickerChoiceListener protocol is already defined for you as well. Your preferences view controller must listen and update the model when a changed color arrives. The color picker does *not* know or care which color is being customized; its only job is to select a color. So, your controller is responsible for knowing where the color needs to get stored by remembering which button was tapped in the first place.

Are you starting to see how these MVC patterns repeat again and again at different scales?

1. (30 pts)

Connect the preferences view controller to the game view controller: first, embed the main controller in a UINavigationController. Then, add a **Preferences** button on the main UI. Draw a **show** segue from this button on the game controller to preferences controller. Add a **Dismiss** button on the preferences side that gets you back to the game. Naturally test this transition back and forth right away.

1. (30 pts)

You probably noticed that all this effort is pretty unsatisfying because the preferences controller is using a private model that has nothing to do with the version stored in the main game controller. Fix this: in prepareForSegue in the game controller, grab the preferences controller and set its model. This means it’s time to change its type to Optional since there’s no meaningful model until the game controller decides to share it. Ensure that everything really works! Picking a color for the player should propagate to the Preferences controller using the ColorPickerChoiceListener subscription, then that chosen color should propagate to the Game view through the GamePrefListener subscription!

**PRETTY COOL EH?**

1. (25 pts) Sadly, the preferences are still lost when the App is quit. After all these development cycles, wouldn’t it be nice if the most recent changes were not lost on every launch? Fix this by implementing a UserDefaults key/value storage every time a preferences model value is changed. **Use the design pattern shown in the reading** to the extent that you use an enum case to represent each key.
   * This model needs to have an init that reads in the user defaults, and falls back gracefully to defaults if any are missing.
   * Using property observers, *save out* the updated values every time they change.
   * Test the whole setup: when you simply instatiate the preferences model from the main controller, it should self-populate from the defaults storage and influence the display correctly!
2. (15 pts)

Come up with your own preference and implement it fully like the others. Anything visible qualifies. Easy ideas (full credit) include the background color of the main game, or a game title label (editable of course in the prefs screen). Intermediate ideas include *counting* the number of fully played games and reflecting that in the display (which should start at zero and feature a **Reset** control in preferences). More advanced ideas include a slider (or pair) that changes the grid size (but only when a game is not underway).

1. (40 pts)

Also sadly, the game model is also lost when the App is quit. This game model is much more elaborate, and can grow much larger and more complex than the four preferences values. Instead of cramming this into UserDefaults, make the game data model conform to NSCoding and encode each field that needs to be saved in the encode:with: required method.

1. (20 pts)

Connect a **Save Game** button to a method that uses the supplied file manager library code to calculate the storage path and store the object. Just emulate the lecture code here.

1. (15 pts)

On startup, restore the game model if it can be found by building the model object out of the unarchiving process, again emulating the lecture code. If it cannot be done, gracefully fall back to an empty game board.

1. (30 pts)

Revel in the newfound Happiness of the App. And have good code style, which we’re still checking on.

**ALTERNATIVE** (“Style Amnesty”). Make sure we know in comments at the top of the game view controller. To avoid any possible style deductions, implement another substantial feature such as actual tic-tac-toe game logic that detects a winner and stops the game, or a timer that limits to a fixed interval the amount of time each user can use to make one turn, or find a use for the tabbed view controller, such as a help screen, and embed the whole App in it. (See the Timer code in TabbedColorView project in lecture 7.) Then throw in some magic constants and bad indentation just for show.