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

**Fall 2018**

**Assignment 3**

**Issued: 09/18/2018**

NO LONGER DUE 09/25/2018

GRAD Students: Assignments 3 (below, 300 pts) and Assignment 4 (to come, 75-100 points) will be due TOGETHER on 2018-10-02. Assignment 4 be a lot smaller in scope in proportion to above point values. It will likely involve adding a component to the picker to separate the food into categories (Food, Drink), as well as the ability to swipe and remove items from the Bill using the Table view.

UNDERGRAD Students: Assignment 3 (below, 300 pts) is due on 2018-10-02. Use the remainder of this week to get caught up on earlier work, syntax, and concepts.

**Purpose:** Become familiar with the cycle of a genuine iOS development in Xcode. Create a working, interactive App. Continue practicing with Swift especially protocols. Learn Storyboarding: AutoLayout, Actions, and Outlets to connect interesting UIViews (visual elements) with actual App data. Begin sophisticated data interactions using delegates and action handling functions.

**Scoring:** Out of 300 points (Undergraduate); 375 (Graduate)

**Readings**

* [Apple Human Interface Guidelines](https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/loading/) Set aside an hour. It will provide a lifetime of value. Skip or skim MacOS-only and advanced topics if you prefer as they will not be covered in the course: 3D touch; System Capabilities (all subtopics); Accessibility; Near Field Communication; Extensions. However, pay particular attention to the following, as they will be part of the final project:
  + Interface essentials: Bars, Views, and Controls
  + Visual Design
  + Modality
  + Navigation
  + Settings
  + User Interaction: Data Entry, Gestures
* Conceptual, Reference: [Delegates and Datasources](https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html) Unfortunately Apple has left this in Objective-C. Focus on the prose.
* Conceptual, Tutorial: [AppCoda tutorial](https://www.appcoda.com/swift-delegate/) shows a detailed real Swift example, keeping in mind that that built-in UIViews like UIPickerView are already built without having to design your own protocol or implement a custom delegating object (LogoDownloader in the tutorial). Also, ignore the network and dispatch queue code. However, this is still very valuable because it gives you some insight into *how* Apple designs its many delegate-using classes like UIPickerView. The aspects of *adopting* the protocol, *implementing* the delegate methods, and *setting* the delegate relationship are still directly relevant.
* UIPickerView: [Apple Reference](https://developer.apple.com/documentation/uikit/uipickerview) Note the closely related datasource and delegate protocols near the top. [PickerView tutorial](https://www.ioscreator.com/tutorials/picker-view-ios-tutorial-ios10) Note in all online tutorials, the delegate and datasource are set in viewDidLoad which is a cluttered style, as this method tends to grow out of control. In E65g, you are required to do so from Storyboard settings, or in the didSet observer of the relevant UIPickerView property of your containing UIViewController.
* [UITableView reference](https://developer.apple.com/documentation/uikit/uitableview) Tables can get very complex. Focus only on the fields and methods used in this assignment:
  + [UITableView.dataSource property](https://developer.apple.com/documentation/uikit/uitableview/1614955-datasource)
  + [UITableView.delegate property](https://developer.apple.com/documentation/uikit/uitableview/1614894-delegate)
  + [UITableViewDataSource.numberOfSections](https://developer.apple.com/documentation/uikit/uitableviewdatasource/1614860-numberofsections)
  + [UITableViewDataSource.numberOfRowsInSection](https://developer.apple.com/documentation/uikit/uitableviewdatasource/1614931-tableview)
  + [UITableViewDataSource.cellForRowAt](https://developer.apple.com/documentation/uikit/uitableviewdatasource/1614861-tableview) This one is the most advanced. For this assignment, rely on the supplied boilerplate pattern to set String data.
  + [UITableViewDelegate.didSelectRowAt](https://developer.apple.com/documentation/uikit/uitableviewdelegate/1614877-tableview)
* Autolayout: [Apple Conceptual discussion](https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/AutolayoutPG/index.html) Here, skip over **Auto Layout without Constraints**. We will not be using StackViews. (Although they’re useful and in fact less work for some scenarios, it’s important to master constraints first.) Some topics in **Advanced Auto Layout** may be covered near the end of the course but not for this assignment.

**Project Structure:** Create an iPhone app that implements a visual interface to the RestaurantBill and RestaurentMenu class (together, comprising your data model). Start a Single-View iOS App Xcode project. Use StoryBoard, not code, to create, constrain, and connect the UI.

**Problems**

1. Pick up the [assignment 3 repository](https://classroom.github.com/assignment-invitations/1a62f56e8a061101ecd2f050dac5d543).
2. Create a **Single-View iOS App** with a project name Assignment3-*githubusername* (substitute your actual GitHub username). The bundle identifier must beedu.harvard.cscie65g.Assignment3-*githubusername*.
3. Create 3 new files: Shared.swift, RestaurantBill.swift, and RestaurantMenu.swift. Copy the relevant class code to the two class files. Put the TippingLevel enum, the MenuProtocol from Assignment 2, a new BillProtocol built the same way (just the essential functionality extracted) and centsToUSDollars conversion code from Assignment 1, in Shared.swift. Note that in the long term, it’s important for protocol code and implementation code to go in separate files.
4. Rename the default UIViewController class *and* file name to BillViewController. Remove the didReceiveMemoryWarning boilerplate.
5. **Layout** The overall goal is to base a small App around the ability to manage a Restaurant Bill, that is, bring a visual interface to the RestaurantBill.addItem()function from Assignment 2. All vertical and horizontal adjacencies should use the **Standard** spacing.

Xcode 9 has introduced a new class of warnings called **Localization Contraints**. These can be ignored for now and added as we gain sophistication in layout.

To vertically align adjacent UILabels and UITextfields, use the **Last Baseline**, not bottom or center. This is the right way to think about text which may be in different sizes and fonts.

Money fields should look "good" and not jump around at all when values change. In particular the decimal point should never shift left or right. To this end, always use a monospace font, always show dollars and cents, always right-align the text internally, and always make the field wide enough to hold the highest imaginable value. Finally, never combine fixed labeling (explaining what something is) with the actual value into a single view (label or text field). Finally, vertically stacked money fields should have their decimal points horizontally aligned (consider a spreadsheet column of monetary values and the regularity you would expect there).

When anchoring views to the superview, Xcode now calls this the **Safe Area** to indicate that the edge used is still safe from interfering with the status bar. When dragging constraints in the hierachical view, use the **View** node *above* the **Safe Area** node to get all the options. Any time leading or trailing margin (left or right edge) to the root view is used, Xcode uses a 16-point distance. If you ever manually edit constraints and lose this value, manually type in the 16 in the constraint editor for the **Constant** to keep margins consistent.

If you are sufficiently sophisticated, you may use blank generic UIViews as containers to create intermediate layout regions that have the desired effect. For example, you can group all of the bill summary information in a UIView, and constrain the whole grouping to the main layout. This is not necessary: if there are four labels in a row above which the table must sit, you can just set the vertical spacing constraint between the UITableView and the leftmost UILabel.

* 1. Enter **Main.storyboard** from the Project browser.
  2. With the **View Controller** selected, within the **Identity Inspector** at right, change identity from (the now non-existent) **ViewController** to the updated name **BillViewController**.
  3. In the root view of your BillViewController, lay out a UILabel on top, horizontally centered, titled **Your Bill**. Make the font 2 points larger than the default. For this and all further layout instructions, you must use Storyboard constraints.
  4. Directly beneath, layout a UIPickerView at the default height and extending from Leading to Trailing margins of the superview.
  5. Implement an item-adding interface: a UIStepper anchored to the left margin, a UILabel to its right with the label **Quantity**, and a a UILabel to its right that shows the current value. Finally a UIButton should be trailing-margin anchored with the text **Add to bill**. The **Quantity** text label should share a vertical center with the stepper; the actual numeric quantity UILabel should be baseline-aligned to the **Quantity** label. The button should be vertically centered with the stepper.
  6. Implement a pricing interface: Make two adjacent UILabels. The first one is left-margin anchored and directly beneath the stepper, labeled **Line item price:**. The second is the actual price. It should be in a monospace font and right-aligned internally. It should be baseline aligned to its explanatory label, and leading-edgealigned to the quantity actual-value UILabel just above. It should have an explicit width large enough to contain the text $10000.00.
  7. Implement a summary interface: Anchoring from the *bottom*, show a cost summary, laid out however you see fit, but all clearly labeled, as a series of UILabels. All money fields should be consistent (remember the *consistency* principle from Apple guidelines?) with the right-aligned monospace format used for the line item total field.
     1. The food total
     2. The tip
     3. The tax
     4. The final total
  8. Implement a bill content interface (this feedback only, no interaction for now): Lay out a UITableView. It should extend from leading to trailing margin of the superview, and extend vertically from the bottom of the pricing interface to the top of the cost summary.

1. **Configure the table** to allow individual cells to be constructed from code.
   1. In the attributes inspector for the table, change the number of **Prototype Cells** to 1. This means there will be one style of cell (and as many rows in the table as you want, dynamically — note the **Content** is set to **Dynamic Prototypes**.
   2. Change the **Style** of the UITableViewCell (that just got created underneath the UITableView node in the View hierarchy) to **Subtitle**. This style comes pre-built with two formatted text areas, one smaller and capable of holding detail information.
   3. In the view hierarchy, the table node will gain a child node: a **Table View Cell**. Select it, and in its attributes inspector, assign the Identifier to something meaningful and unique for the forseeable future of the App which might have many tables. (So **tableCellID** or **HelensTableCell** are bad choices!).
2. **Make Connections** from Storyboard to code
   1. Open the code side-by-side using the **Assistant editor** button.
   2. Create Outlets for all views that will require updating or querying based on interacting with other views: all money fields, the quantity field, and the stepper.
   3. Create Actions for all interactive views: the stepper and the add-item button. You can leave them empty for now, but debug print statements and breakpoints are an excellent idea.
3. **Prepare the data model** for its new interactive UI demands. The UI needs to display these objects in a list, and so needs to know the ordering. However, we now have a minor conflict. Dictionaries are unconcerned with ordering, only key lookups. We will take the least user-friendly, least computationally efficient, but most programmer-friendly, path here.

As the software matures you would transition to sensible approaches involve sorting and caching.

You will need to augment both classes and the related protocols.

* 1. Each needs to supply a count itemCount (as computed property): Number of items in the menu, and number of line items in a bill.
  2. Each needs to supply an item given a numeric index. Since you probably stored the items in a Dictionary in both, you’ll need a function:  
     func item(at index: Int) -> MenuItem and func item(at index: int) -> (description: String, quantity: Int) that internally and temporarily transforms the values into an Array for integer indexing:   
     let asArray = Array(myDictionary)  
     Now you can easily work with integer indices.
  3. Now that the UI demands the tax and tip breakdown, these should become official read-only Int values of the protocol and implemented straight-forwardly.

1. **Own the data model and sync to it** in the view controller
   1. Add a new method addTestData() -> Void to the BillProtocol. Note it takes no arguments and returns nothing, so it is a pure mutator. Implement it in the concrete type using a private static array of test data copied into instance storage. This helps us test the UI (the table and monetary values, in particular) immediately without tediously adding multiple items every time the App starts, also with the same data every time for consistency against an evolving UI. Call this method in the view controller, right after the model is constructed. (Mild extra credit idea: it would be nice if this was interactively controlled. A real testing harness would tie this to an unobtrusive button right in the UI, rather than assuming it should be done on startup.)
   2. Declare and instantiate the RestaurantBill and RestaurantModel instances as stored properties in the view controller. This will become clearer as time goes on, but you must *declare the type to be the protocol type* yet *call the initializer for the actual type*. Because you do not have access to the init method of the view controller, there are two ways to initialize a such a property:
      1. Right in the declaration, which is really a backdoor into the initializer. However, if it takes multiple statements to perform the initialization, you will need to use an [immediately-evaluated anonymous closure](https://encyclopediaofdaniel.com/blog/swift-immediately-invoked-closures/). (Note: despite my namesake, that is not my [blag](https://www.explainxkcd.com/wiki/index.php/148:_Mispronouncing).) The advantage of this way is we do not have to use an Optional type. The other advantage is we can declare it as a let instead of a var if we want to protect against accidental future reassignment.
      2. In viewDidLoad(). This means for initialization to work, the stored properties must be declared as Optional. This relieves you of having to get comfortable with the anonymous closure technique above, but the model must be carefully unwrapped once for every method that needs it. (Side note: Most online tutorials do it this way, but they also take the cheap shortcut of declaring an implicitly unwrapped Optional type and never checking it.)
   3. Write a new function updateUI that will eventually update *all* model-dependent views. For now, it should just update all four cost summary views, which will come directly from the RestaurantBill instance. A debug print is very good here too. Here, you’re translating model data into Strings, and simply setting the text of the views to these String values. This function will be called when anything about the model changes. This is where currency formatting comes in very handy.
2. **Adopt the delegate protocols** in the view controller to handle the demands of the complex picker and table views.
   1. **Using separate extensions for each**, declare the view controller to adopt the delegate and datasource classes for both the picker and the table. This will cause a few compiler errors to pop up but will also prepare the auto-complete to offer the included methods for easy stub method creation. (For example if you type **numbe**you’ll get the UITableViewDataSource.numberOfRowsInSection method stub.)
   2. Implement the 3 table view datasource methods mentioned above. The data comes from the RestaurantBill portion of the data model. You can leave these methods as stubs (returning hard-coded data) if you are not yet comfortable wiring them to the data model, and come back later. The minimum stub for the cellForRowmethod is:  
      let cell = tableView.dequeueReusableCell(withIdentifier: "*chosen-identifier*", for: indexPath) return cell // will show up empty but valid where *chosen-identifier* is the Storyboard Identifier chosen during table configuration above. Note cells are special objects and they should *never* be instantiated directly. You must use the factory method shown.
   3. Implement the 1 delegate method. Leave this as a stub for now with a debug print.
   4. Implement the necessary picker datasource methods. There are 3 exactly analogous to the table view. Instead of UITableViewDataSource.numberOfSectionsthough, it’s UIPickerView.numberOfComponents, and instead of the complicated cellForRow method, it’s the much easier titleForRow method. The picker view data comes from the RestaurantMenu portion of the model, not the bill!
   5. Implement the 1 picker delegate method, exactly analagous to the table. Leave this as a stub as well.
   6. Sign up the view controller to play these 4 critical delegation roles. In Storyboard, assign both the delegate and datasource properties of both the picker and the table view to the bill view controller. This is the one time to use the yellow circular receiving icon of the View Controller object. The assignment can be done in code instead, but not for this assignment.
3. **Verify the layout** Run the project in a standard (non-plus) sized iPhone 8. Ensure the runtime layout matches your expectations and fix constraints and runtime warnings.
4. **Verify the read-only direction** where data from the data model flows into the view. Ensure that the hard-coded menu displays in the picker view.
5. **Verify the action wiring** to ensure all interactive events are indeed captured, before acting on them. Intermediate output values (the text labels) will not yet be correct.
   1. Ensure that rotating the picker triggers its delegate handler didSelectRow, and that you can debug-output the current selected row.
   2. Ensure the add-item button triggers its action handler.
   3. Ensure the stepper triggers its action handler.
6. **Complete the action handlers**. All the infrastructure is present now for the interactive side!
   1. Complete a missing portion of updateUI: the quantity display should retrieve its value from the stepper.
   2. The stepper action handler should simply call updateUI (which will now, among other things, consult the stepper’s current value and reflect it in the related label.)
   3. Create an outlet (stored property in view controller) for the picker view. We need to be able to consult it outside of the context of its datasource and delegate methods.
   4. Complete another missing portion of updateUI: [retrieve the row](https://developer.apple.com/documentation/uikit/uipickerview/1614369-selectedrow) from the picker outlet, correlate with the data model, and update the line item display accordingly. Now, the picker row selection handler can simply call updateUI.
7. **Implement table updates**.

You will likely have noticed that adding a new line item to the bill does not add anything in visual bill table. This is because tables need an explicit signal to know that there is new (or different) data waiting in the data model. The delegate / protocol design is inherently *passive*: views know *how* to ask for data, but not *when*, especially as that data changes during App interaction. (The first time they are laid out and told to draw themselves, during the willAppear portion of their controller’s cycle, they do in fact call upon their data sources, which is why you get anything at all on startup. But not after.)

There are no easy around the annoying, tedious, and troublesome process of making sure your complex views stay up to date with your complex data.

To force a table to rebuild itself completely (the easiest update method), we must call [reloadData](https://developer.apple.com/documentation/uikit/uitableview/1614862-reloaddata) as soon as we become aware the table data has changed. For now we’ll take the simplest shortcut:

* 1. Create an outlet for the tableView. We need to be able to update it outside of the context of a datasource or delegate method (where it’s passed in as a parameter.)
  2. In the action handler for the add-item button, in addition to calling updateUI call the reloadData method on the table view outlet.

1. **Set up initialization** Non-complex views like UILabels do *not* have a nice delegate setup to know what their initial values should be. So we have a different tedious annoyance for these. (If I designed a UI system, I would make the view catalog consistent: every single subclass would have a datasource and delegate, even if it was a single black-and-white pixel.)

This is the job of viewDidLoad(), which gets called after all necessary initialization and plumbing (all views in Storyboard are initialized and connected via the Outlets and Actions). Starting up an App is a user-event just like tapping a model-mutating button, and our responsibilities are no different: we must ensure the entire view is current with respect to the data. So we must call updateUI here. Required to answer in comments right above the call: What was missing before we did this?

**Finalized Grading breakdown**

* Data model classes (RestaurantBill and RestaurantMenu implementations) each get their own separate file. Each should have no references to UIKit. The BillViewController should rely on these classes for all actual data: not even dummy or test data should be declared or initialized in the view controller. We will not accept submission otherwise. The easy way to enforce clean data model files is simply not to import UIKit in those files. (import Foundation instead)
* 50 Inter-item layout with plausible alignments, no conflicts or ambiguities
* 15 Internal item formatting as specified (money values)
* 20 Initial state correct (empty / zeros)
* 30 Menu model mapped to picker view
* 20 Picker functionality: updates line item right away
* 30 Bill model mapped to table view
* 15 Stepper functionality
* 35 Add Item functionality, bill listing and all money values update right away
* 5 No UI element allows invalid inputs, such as negative values
* 10 No crashes, generally during our testing
* 15 All UI updates consolidated into updateUI, not action handlers
* 10 Protocols correctly isolated and contain all functionality
* 45 Other general code style