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

**Fall 2018**

**Assignment 5**

**Issued: 10/04/2018 Due: 10/09/2018**

**Scoring**

Undergraduate: 145 points; Graduate: 200 points

**Purpose**

Learn and harness the full MVC pattern by analyzing and enhancing the in-class MVC example.

**Overview**

Because of what has been covered so far in lecture (and not covered), and to build understanding brick-by-brick, the former Assignment 5 material has been moved to Assignment 6 and this is the new Assignment. MVC feels quite complicated at first and requires working within existing code to get comfortable. If you are able to move fast you may certainly start on Assignment 6 as well but will have to rely on Internet tutorials for the gesture recognition and UIBezierPath rectangle rendering techniques.

There are a series of written questions to ensure you fully understand the supplied code. Then there are several enhancements to be made involving coding and Storyboard. Start off by making a complete copy of lecture05/CalculatorMVC using the same Project-copying technique used before. Take the time to update the Project Name and BundleID to include your git username, and ensure it runs. Now you can freely experiment to help answer the questions, and have a starting off point for the enhancements.

Pick up the [Assignment 5 repository](https://classroom.github.com/a/MsQjHKtD).

**Written questions**

For these, it will be important to keep in mind there are *two* user interfaces for entering numbers and triggering the answer, and that they (should) stay in sync. You can answer in a separate RTF, TXT, or HTML document instead of cramming things into comments. No PDF, Word, Pages, or other binary formats please. You can just call it answers.txt at top level of repository. The first batch of questions deals entirely with the model. Optimization questions require a more holistic view. The last questions are centered around the controller but require a full App understanding.

1. **Optionality**
   * (3) Why is \_answer an Optional?
   * (4) Look at answerAsText. The most common use of ?? is just:  
     let alwaysNonOptional = optionalValue ?? defaultValue  
     Why does the left side of computation for answerAsText require the extra part compared to the above? Why does the right side require a nested ?? expression?
   * (3) Why is the \_dataListener an Optional?
2. **Error-checking**
   * (3) Why does the divide operation require special handling? Do we still need that if the entire model used Doubles instead of Ints?
   * (5) Why is there conditional logic for getOperandIndex?
   * (2) ArithmeticError adopts Error because it feels natural, but this is not necessary. Under what conditions would it become necessary?
3. **Generality**
   * (6) Your colleague comes along and says “Impressive model. My UI needs exactly this but it uses UInt16 (unsigned 16-bit integers). Too bad yours is hardcoded to type Int!” What mechanism does Swift have that would allow us to *generalize* the model code and *not* assume the underlying data type?  
     Hint: You are already using this mechanism all the time, but syntactic shortcuts hide it. What does an Array declaration look like with the syntactic shortcut removed? See [the collection types](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html) chapter for a quick review but don’t get caught up in other details. This should take less than 10 minutes of research time.
   * (3) There’s a comment that notes a small coding shortcut as a “beginner design error”. After completing the code modification exercises, what modification showed that to be true?
4. **Optimization**
   * (8) Looking over the model code, you note that the set implementations trigger computation, which triggers other things. You want to filter out unnecessary work so if the caller is setting the value to what is already stored, you want to optimize that away. Implement this for leftOperand.
   * (10) Ever on the lookout for optimizations, you want to do the same for the didSet observer of \_answer. Why does this *not* work? If you can figure it out without writing code, you can answer purely in writing, but thorough UI testing will show the way.
   * (7) Now suspicious of optimizations, you note that setEntireOperation isn’t strictly necessary and bloats the Protocol. What could a caller do instead? Explain the downside, or more neutrally, the consequences, when removing this method and using the remaining alternative. Take care to look at the external (user visible) behavior and internal (efficiency of computation and code clarity).
5. GRADUATE CREDIT ONLY  
   **Cool stuff in Swift 4.2**  
   (5) Since we were forced to upgrade to Xcode 10 / Swift 4.2, what is this CaseIterable all about? That is, what is more convenient now and how do we leverage this in the program?
6. **UI synchronizing**
   * (5) You know that updateUI has to be called when the model indicates values have changed. Why does it appear in viewDidLoad?
   * (10) Looking over didSelectRow, you note that it ends immediately after making a single assignment (with a nested lookup). What’s the chain of events that actually make the UI update? The granularity should be at the level of method call. Property-related code like observers and explicit set and get blocks count as methods.
   * (6) You note that in updateUI, the operationPickerView is explicitly spun to the stored values. This seems wasteful if you just spun the picker to change a value. Why is it necessary?

**Code modification problems**

To continue to enhance your understanding of MVC, start with this working infrastructure and add to it, as preparation for creating your own.

1. **Cosmetic**
   * (4) Numbers often look better in a monospaced font and right-aligned. Do this for the input text fields and the answer label.
   * (6) The history output is a little funny-looking. Fix it. Keep the code elegant. Hint: what is a helpful computed property in the operator enum type?
   * (5) The picker wheels look awkward with so much whitespace. You remember that while some delegate methods of the picker report interesting events, other optional methods customize the appearance of the view. Find the one that squeezes them together horizontally; the exact amount is whatever looks good to you.
   * (10) Tweak the table appearance for practice. Give it a subtle but noticeable background color in Storyboard. For a consistent background, you will need to work with the UITableView, the contentView of the cell, *and* one more thing. This is a lot easier in Storyboard but in code is OK if you want to teach yourself. Make the rows a bit shorter – this must be in code.
2. **New Features**
   * (10) Flesh out the operations in the top area: add multiply and divide functionality.
   * (15) Add an exponentiation operator, using the symbol ^. Several layers of the code will need updating, not just the controller. Ensure it appears in the picker and as a button. There are some math-y special cases and to detect as you move from floating-point back to integer values. You can modify ArithmeticError.
   * (20) The table has nice output, but selecting rows is annoyingly inoperative. Capture selection events and configure the UI to recapture that (so special) moment in time. That is, sync the picker wheels, text fields, and answer field to the selected history operation. **Note**: The model is not set up for this. You will need to re-conceive the history representation, which is frankly a bit lazy right now with just String to hold each entry. Define a struct that natively represents an operation. You have done this already for MenuItem so use everything you learned from that.
   * GRADUATE CREDIT ONLY  
     (10) Add a working **clear history** button, placed whereever you see fit. Ensure that only one update message is sent by the model.
3. **Current features: Boundary conditions**
   * GRADUATE CREDIT ONLY  
     (25) The pickers currently spin to the highest or lowest value if a typed-in value is out of range. Add special “Too big!” and “Too small!” values at the end of the ranges in both numeric spinners. They should be used to handle the out-of-range problem. Since the model represents a range of integers, the logic can remain in the Controller although the Controller will certainly need to consult the model to do this. (This is where the principles of MVC require a judgement call; it’s possible to keep all changes in the model and make the model a lot more complicated. It could conceivably represent string *and* integer values – consider enum MultiVal { case String(stringVal); case Int(intVal); } – at the cost of some ugly internal code. So, what we claim is that these special values are really a user interface issue, logically distinct from the model. Pretty much like separate labels, even though they happen to be embedded right in the picker wheel. Any thoughts?)
   * GRADUATE CREDIT ONLY  
     (15) Enhancements often cause new wrinkles. The values “Too big!” and “Too small!” are *output* values only. If you were a really good tester (were you??) you tried to spin to these values to see what happens. If the user spins to either of these, quietly and politely detect that case, and spin back to the *nearest* (not previously selected) valid value, before computing anything.