# CSCI 448 – Lab 04B
Friday, February 09, 2023
LAB IS DUE BY **Thursday, February 23, 2023 11:59 PM**!!

The LGG Dev Team has been receiving lots of praise now that the Quizler app supports multi-answer questions. Users are really enjoying the array of questions they can cycle through. However, another complaint has started to be heard frequently – users can answer questions multiple times and continue to increase their score!

The Dev Team decides they'll restrict each question to only be able to be answered a single time.

## Step 1 – Only One Chance

You volunteer to start the refactoring process for the View Model to track if the question has been answered or not. You realize this will only involve modifying the View Model and View components of the app.

### Part 1.I – Track the Question Status State

Since the View Model is starting to become slightly more complex, begin by adding a new package `.presentation.viewmodel`. Move the `QuestionViewModel` and `QuestionViewModelFactory` into this new package and refactor each to be named `QuizlerViewModel` and `QuizelrViewModelFactory`. You are starting to realize this View Model is doing more than just a single question – it's managing the state for our entire app.

Now your package tree should be:

- `quizler`
  - `data`
    - `Question`
    - `QuestionRepo`
  - `presentation`
    - `question`
      - `QuestionButton.kt`
      - `...`
    - `viewmodel`
      - `QuizlerViewModel.kt`
      - `QuizlerViewModelFactory.kt`

#### Part 1.I.A – Track the State

Inside this new `viewmodel` package, create a new `enum class` called `QuestionStatus`. This enumeration will have three values: (1) `UNANSWERED` (2) `ANSWERED_CORRECT` (3) `ANSWERED_INCORRECT`. We'll use this to know if an answer is answered or not and how it was answered.

In our `QuizlerViewModel` class, we'll add a private `MutableList` of `QuestionStatus`. This list needs to be the same size as our question list and have each element initialized to `UNANSWERED`.

Additionally, add a private mutable state member that will track the current question's status. Initialize this to the status for the current question index.

The only information that needs to be publicly exposed is the status of the current question state. Add a public immutable data member whose getter returns the immutable state of our current question state.

**Part 1.I.B – Update the State**

When the `answeredCorrect()` method is called, we'll update the status for the current question index and set it to be `ANSWERED_CORRECT`.

Add a new method called `answeredIncorrect()`. When called, log it's calling then set the status for the current question index to be `ANSWERED_INCORRECT`.

Finally, when the previous or next methods are called in addition to setting the current question state, also set the current status state.

## Part 1.II – Display the Question Status State

Over on the View side, we'll need to work bottom up to have the ability for a question to be answerable only once.

**Part 1.II.A – Disable a Button**

Begin in `QuestionButton` by adding another parameter to the composable function called `enabled` that is a `Boolean` and set to be `true` by default.

Use this value to set the `enabled` property of the enclosed `Button` composable.

Add a new preview composable that calls your `QuestionButton` with `enabled` set to `false` to preview both button states.

**Part 1.II.B – UI Logic Based on Status State**

We set the parameter to be true by default to not break any existing usages of our `QuestionButton` composable. The `QuestionDisplay` composable is the level that displays the buttons corresponding to our answer choices, these are the ones we wish to disable. Currently this composable receives a parameter corresponding to the current `Question`. Add another parameter that corresponds to the current question status (thus its type should be `QuestionStatus`).

For each of our buttons, we'll set the enabled argument to be the result of if the `questionStatus` equals `UNANSWERED`. If the question has not been answered, then enabled is true. Otherwise, the button will be disabled.

Again, add some previews for each of the question status values.

**Part 1.II.C – Observe the Status State**

The `QuestionScreen` composable receives the `QuizlerViewModel` object and sends the corresponding state down to the appropriate composables. Pass the current question state down to the `QuestionDisplay` composable.

Update the `onWrongAnswer` lambda to call the recently created `answeredIncorrect()` method to trigger a wrong answer.

Build, deploy, run, and answer some questions. It seems this is mostly working, except only after navigating away from a question and then returning to the same question are the buttons disabled.

### Part 1.III – Finish Updating the State

The only time we are modifying the current status state is when the previous or next buttons are pressed. We need to update the current status state when a question in answered. Currently, `answeredCorrect()` and `answeredIncorrect()` are only updating the list of statuses. They each need to also update the current status state to reflect this internal change.

Rebuild, redeploy, rerun, and reanswer some questions. Instant only one try. Choose wisely.

## Step 2 – Show the Result

Trying to get ahead of future feature requests, now that the buttons can be disabled they all appear the same without anyway of knowing did the user previously answer correctly or incorrectly. The View Model is already tracking this state information, we'll need to add in more UI Logic to display it as such.

### Part 2.I – Make New Colors

The file `.ui.theme.Colors.kt` has our theme colors defined. We'll add three new colors. Use the following (or make your own – just make sure they pass the WCAG AA contrast checker for accessibility: https://webaim.org/resources/contrastchecker/):

```
val Gold60 = Color(0xFFD39F10)
val Blue20 = Color(0xFF0C23F0)
val Red40 = Color(0xFF5F1709)
```

(*Note: the colors are specified in ARGB order.*)

### Part 2.II – Expose the Color Property

Inside of the `QuestionButton` composable, expose the `colors` property of the internal `Button` composable.

In other words, add a `colors` parameter to `QuestionButton` that is then passed to the `Button` function call. Be sure to choose the correct data type. Specify the default parameter value to be `ButtonDefaults.buttonColors()` so existing usages are not broken by the added parameter.

Feel free to also expose the `borders` property if you want even more control over the look of the button.

### Part 2.III – Choose the Color Based on Status

In `QuestionDisplay`, we are already using the question status to determine if the button is enabled or not. We will also use the question status to determine the color of the button.

Create variables the will store the color settings for each of the status states. These variables correspond to the following usages of `ButtonDefaults.buttonColors()` with corresponding arguments:

- **`correctButtonColors:`**
  - **`disabledContainerColor = Gold60`**
  - **`disabledContentColor = Blue20`**
- **`incorrectButtonColors:`**
  - **`disabledContainerColor = Red40`**
  - **`disabledContentColor = Gold60`**
- **`defaultButtonColors:`**
  -

The default should use the result of calling `buttonColors()` with no arguments.

We'll now set the color of our QuestionButton based on the status and the correct answer for the question. For each button we'll check:

- If the status is correct and this button is the correct answer
  - Then use `correctButtonColors`
- Else if the status is incorrect and this button is the correct answer
  - Then use `incorrectButtonColors`
- Otherwise
  - Use `defaultButtonColors`

This gives us the ability to always highlight the correct answer. Based on how the question was answered, we'll either color the correct answer in blue text on a gold background or gold text on a red background. Any buttons that don't correspond to the correct answer will display in their default fashion.

Build, deploy, run, answer, and check out the color madness!

## Step 3 – Add Some Flair

You decide to start taking initiative with the View and make the app look a little more polished. You want to give the app a more reactive and interactable feel. You read recently about Material Design and realize Little Green Games is already using material design components – `Card()`, `Button()`, and others. You feel the team should leverage all the abilities of these components. You want to start by adding elevation to cast some shadows.

### Part 3.I – Card Shadow

You're about to start setting a number of elevation properties on the existing `Card` composable inside of `QuestionDisplay()` but quickly see there already exists an `ElevatedCard()` composable. You replace your `Card` with the `ElevatedCard`. Yay, instant shadow.

### Part 3.II – Animated Button Shadow

You then make the same change inside of `QuestionButton()` – replacing `Button()` with `ElevatedButton()`. But you want to do a bit more. You want the user to feel the effects of pressing the button.

The `elevation` property accepts several elevation states and automatically animates between them. Set this property to be the result of calling `ButtonDefaults.elevationButtonElevation()` with the following arguments:

- `defaultElevation : 2.dp`
- `pressedElevation : 8.dp`
- `focusedElevation : 2.dp`
- `hoveredElevation : 2.dp`
- `disabledElevation : 0.dp`

Now, when the button is pressed it will lift up and cast a larger shadow. If the user moves off the button, it will drop down. If a button is disabled and uninteractable, then it will rest on the surface without any shadow.

Build, deploy, run, interact, and admire the new look and feel.

## Step 4 – Deploy Your App

Armed with Material Design knowledge, you set out to put the landscape layout in place before another feature request comes in.

LAB IS DUE BY **Thursday, February 23, 2023 11:59 PM**!!