Criterion C: Development

Product Development

Use of libraries including SwiftUI and SwiftData

SwiftUI provides views, controls, and layout structures for creating the app's user interface ("SwiftUI").

SwiftData enables persistent data storage and provides simple and efficient ways to manage SQLite databases in Swift. It is capable of handling SQL queries and data manipulation tasks (Expert App Devs).

```swift
import SwiftData
import Foundation

//This page will save data inputted by the user on the Budget page
@Model
class Expense {
    @Attribute(.unique) var id: UUID //unique ID for each entry on Budget page
    var name: String
    var cost: Double //takes in integers with decimals
    var dateCreated: Date

    init(
        id: UUID = UUID(),
        name: String,
        cost: Double
    )
    {
        self.id = id
        self.name = name
        self.cost = cost
        self.dateCreated = Date()
    }
}
```

```swift
import SwiftData
import Foundation //Since UUID and Date are part of the Foundation framework, importing Foundation is required to use these types in the Recipe class definition.

//This page will save data inputted by the user on the Recipes page
@Model
class Recipe {
    @Attribute(.unique) var id: UUID
    var name: String
    var ingredients: String
    var lastMade: String
    var tutorialLink: String
    var dateCreated: Date
    var notes: String

    init(
        id: UUID = UUID (),
        name: String,
        ingredients: String,
        lastMade: String,
        tutorialLink: String,
        notes: String
    )
    {
        self.id = id
        self.name = name
        self.ingredients = ingredients
        self.lastMade = lastMade
        self.tutorialLink = tutorialLink
        self.dateCreated = Date()
        self.notes = notes
    }
}
```

Figure 1: Creating two Models, called "recipe" and "Expense"

To use **SwiftData**, I first need to create data models and model containers (in this case "Recipe" and "Expense") (Figure 1). Model() converts a Swift class into a stored model that's managed by SwiftData ("Model()"). **ModelContainer** is for creating and managing the actual database file used for all SwiftData's storage needs (Figure 2), while **ModelContext** is for tracking all objects that have been created, modified, and deleted in memory, so they can all be saved to the model container later (Hudson).

```swift
import SwiftUI
import SwiftData

@main
struct The_Ice_Cream_AppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
                //Telling SwiftData which models to manage
                .modelContainer(for: [Recipe.self, Expense.self])
        }
    }
}
```

Figure 2: Creating modelContainers for Recipe and Expense

"**Context**" in models refers to a data environment or a container that holds data and provides a way to interact with that data. The context can be used to manage how data is accessed, modified, and

persisted by the user inside an app (CodeWithChris). Data is saved on the user's computer, even when the app is quitted.

Home Page

This is structured called Platform with two properties, name and color. Platform is used to create the three views: Recipes, Calculator, and Budget (Figure 3).

```swift
import SwiftUI

struct Platform: Hashable {
    //defining an interface (entrance for text inputs and images)
    var name: String
    let color: Color
}

struct ContentView: View {
    var platforms: [Platform] = [
        .init(name: "Recipes", color: .black ),
        .init(name: "Calculator", color: .black),
        .init(name: "Budget", color:.black )
    ]
```

Figure 3: Defining a Struct called Platform

NavigationStack and **NavigationLink** are used to link three different Views to the Home View: Each view (Recipes, Calculator, and Budget) is stacked on top of the home view. When the View's button is pressed, the View is pushed on top of the home View. When the return button is pressed, the View is popped off of the home View (Figure 4).

The NavigationStacks are based on data, not the views (Prater). By using NavigationLink, the "platforms" can be accessed through the tabs.

```swift
    var body: some View {
        NavigationStack{
            List {
                Section("Let's Make Some Ice Cream!") {
                    //ForEach iterates over the platforms array, where each platform has a name and a color.
                    ForEach(platforms, id:\.name) {platform in
                        //This creates a navigation link to navigate to one of the three pages when tapping on an ice cream platform
                        NavigationLink(value: platform){
                            HStack{
                                Image(systemName: "lightbulb")
                                    .font(.system(size: 70))
                                    .foregroundColor(.black)

                                Text(platform.name)
                                    .foregroundColor(platform.color)
                                    . font(.system(size: 70))
                                    .bold()
                            }
                        }
                    }
```

Figure 4: Use of NavigationStack and NavigationLink on Home page

A **NavigationDestination** modifier is used to define the destination views for different "platforms" within the NavigationLink (Figure 5). Then, a switch statement is used to select one of the three platforms ("Switch Statement").

```
54              .navigationDestination(for: Platform.self) { platform in
55                  //A switch statement is like an advanced if statement
56                  //All possible outcomes are listed, and one be selected
57                  switch platform.name {
58                  case "Recipes":
59                      Recipes()
60                  case "Calculator":
61                      Calculator()
62                  case "Budget":
63                      Budget()
64                  //Validation method: if the selected page doesn't exist, a message "Unknown Page" shows up
65                  default:
66                      Text("Unknown Page")
67                  }
68              }
69          }
70          .navigationTitle("Home")
71      }
72  }
```

Figure 5: Use of NavigationDestination to access the three pages

Recipe Page

@Query is used to fetch data regarding Recipes from the database. The data fetched automatically reflects updates made to it.

```
8   import SwiftUI
9   import SwiftData
10
11  struct Recipes: View {
12      // @Query retrieves all Recipes from database and then sorts them
            according to "dateCreated" (ordered from new to old)
13      @Query(sort: \Recipe.dateCreated, order: .reverse)
14
15      //Storing all recipe data in an 1 Dimensional array called Recipe (arrays
            in Swift are dynamic by default)
16      private var allRecipes: [Recipe]
17
18      //This controls when the "Add a Recipe" sheet appears. It's hidden at
            first (and only comes on when a button is pressed, which is coded
            below)
19      //states means that when it's updated, we can take actions with it
20      @State private var showSheet = false
21
22      //When a recipe needs to be edited
23      @State private var recipeToEdit: Recipe?
```

Figure 6: Use of @Query to retrieve data from database

ForEach is a view that creates an array of views from an underlying collection of data. It is like a map function that turns an array of data into multiple views. The resulting views can be used within other container views, such as VStack, HStack, and List.

List is a container view that displays a collection of views as rows in a column, like a list. It has many build-in appearance features that the programmer can configure using view modifiers, such as

list styles, headers, footers, sections, and separators. It also has many built-in user interactions like selecting, adding, deleting, and reordering." ("List and ForEach").
Usage: Here, the list is populated with dynamic data using ForEach to iterate over a collection of data and display it in the list (Figure 7).

An **If Statement** is used so that the link is only displayed if the user inputted a link for a specific recipe. If not, a message is shown (Figure 7).

```
35              //Displaying "allRecipes" from the database, in the form of a "list"
36              List {
37                  Section {
38                      //ForEach turns an array of data into multiple views
39                      ForEach(allRecipes) {recipe in
40                          VStack(alignment: .leading, spacing: 6) {
41                              Text(recipe.name)
42                                  .font(.title)
43                                  .fontWeight(.semibold)
44                              Text("Ingredients: \(recipe.ingredients)")
45                                  .font(.title2)
46                                  .foregroundColor(.secondary)
47                                  .textSelection(.enabled)
48                              Text("Last Made: \(recipe.lastMade)")
49                                  .font(.title2)
50                                  .foregroundColor(.secondary)
51                              //If statement to retrieve link saved earlier (only if the user entered a link)
52                              if (!recipe.tutorialLink.isEmpty)
53                              {
54                                  //turns user input into a link
55                                  Link("Link to Tutorial", destination: URL(string: "\(recipe.tutorialLink)")!)
56                                      .font(.title2)
57                              }
58                              else //If user didn't enter a link, this message is displayed
59                              {
60                                  Text("No link was added")
61                                      .font(.title2)
62                                      .foregroundColor(.secondary)
63                              }
64                              Text("Notes: \(recipe.notes)")
65                                  .font(.title2)
66                                  .foregroundColor(.secondary)
67                          }
68                          //When this section of the list is tapped/clicked, the edit recipe sheet shows up
69                          .onTapGesture {
70                              recipeToEdit = recipe
71                          }
```

Figure 7: Use of List and If Statement in constructing scrollable recipe list (iOS Academy)

An AddRecipeView is created, with modelContext to input the new data, with reference the model Recipe:

```
118 //A new view used for the "Add Recipe" sheet, which uses SwiftData
119 //Following client requirements:
120 //Texts for name, ingredients, notes, and links to videos/online tutorials
121
122 struct AddRecipeView: View {
123     //Get SwiftData's Context (used to input new data)
124     @Environment(\.modelContext) private var context
125     @Environment(\.dismiss) private var dismiss
126
127     //State variables because they store the data and track changes in the data
128     @State private var name = ""
129     @State private var ingredients = ""
130     @State private var lastMade = ""
131     @State private var tutorialLink = ""
132     @State private var notes = ""
```

Figure 8: Constructing the AddRecipeView

Encapsulation: By using access control modifiers, such as **private**, when creating variables and functions, I can restricts access to them within the defined scope or file (Odedra).

State versus Binding Variables:
@State facilitates changes to the data and view of that data (while owning the data).

@Binding allows changes in another view (without owning the data, but just a pointer to the data from this view) and if the data in that view is updated, the change is reflected in the original view too ("Driving Changes").

Thus, @State is used in creating the AddRecipeView while Binding is used to create the EditRecipeView (to change the data both within the EditRecipeSheet and on the Recipe page) (Figure 8&9).

```
//This sheet is called when the user clicks on the texts of the entered data. It allows the user to edit their responses and automatically saves the new input values to SwiftData.
struct EditRecipeSheet: View {
    @Environment(\.dismiss) private var dismiss
    @Bindable var recipe: Recipe

    var body: some View {
        NavigationStack {
            Form {
                TextField("Name 🍦", text: $recipe.name)
                    .font(.title2)
                TextField("Ingredients 🥛", text: $recipe.ingredients)
                    .font(.title2)
                TextField("Last Made ⏰", text: $recipe.lastMade)
                    .font(.title2)
                TextField("Tutorial Link 🔗",text: $recipe.tutorialLink)
                    .font(.title2)
                TextField("Notes 📝", text: $recipe.notes)
                    .font(.title2)
                    .lineLimit(10)
            }
            .padding()
            .navigationTitle("Edit Recipe")
            .toolbar {
                ToolbarItem() {
                    Button("Done") {
                        dismiss()
                    }
                }
            }
            .background(Color(red: 0.72, green: 0.9, blue: 0.66).edgesIgnoringSafeArea(.all))
        }
    }
}
```

Figure 9: Constructing the EditRecipeView (Allen)

These are reusable functions for adding and deleting recipes (they insert and deleting data from a sequential file without reading the entire file into RAM):

```
    //This function saves the inputted data for new recipe
    func addRecipe() {
        let newRecipe = Recipe (
            name: name,
            ingredients: ingredients,
            lastMade: lastMade,
            tutorialLink: tutorialLink,
            notes: notes
        )
        context.insert(newRecipe)
        dismiss()
    }
```

```
    //This is a function for deleting recipes
    private func deleteRecipe(offsets: IndexSet) {
        for index in offsets {
            let recipe = allRecipes[index]
            // SwiftData has built-in modelContext
            if let context = recipe.modelContext {
                context.delete(recipe)
            }
        }
    }
```

Figure 10: Creating functions to delete and save recipes

Budget Page

Creating an array (called “Expense”) of items to be stored in the database through the Expense model:

```swift
import SwiftUI
import SwiftData

struct Budget: View {
    // 1) Reads data from "Expense" database
    // "@Query" is used to retrieve data from database
    @Query(sort: \Expense.dateCreated, order: .reverse)
    private var expenses: [Expense]

    // 2) Controls when an "Add Expense" sheet pops out
    @State private var showSheet = false

    // 3) Calculates the total of all expenses
    private var totalSaving: Double {
        expenses.reduce(0) {$0 + $1.cost}
    }
```

Figure 11: Creating the Budget View

Creating a **sheet** that allows the client to modify/add income or expenses to the list:

```swift
    var body: some View {
        NavigationStack {
            //Creates text boxes for user input
            Form {
                TextField("Name 🍦", text: $name)
                    .font(.title2)
                TextField("Ingredients 🥛", text: $ingredients)
                    .font(.title2)
                TextField("Last Made ⏰", text: $lastMade)
                    .font(.title2)
                TextField("Tutorial Link 🔗",text: $tutorialLink)
                    .font(.title2)
                TextField("Notes 📝", text: $notes)
                    .font(.title2)
                    .lineLimit(10)
            }
            .padding()
            .navigationTitle("Add a Recipe") //Title of this sheet
            .toolbar {
                ToolbarItem() {
                    //"Cancel" button for sheet
                    Button("Cancel") {
                        dismiss()
                    }
                }
                ToolbarItem() {
                    //"Save" button for sheet
                    Button("Save") {
                        addRecipe()
                    }
                    //User can only save a Recipe if the Recipe name text box is NOT empty
                    .disabled(name.isEmpty)
                }
            }
            .background(Color(red: 0.72, green: 0.9, blue: 0.66).edgesIgnoringSafeArea(.all))
        }
    }
```

Figure 12: Creating AddBudget Sheet

If Statement: If Expense was selected, amount is negative (subtracting from totalSavings); Else, if Income was selected, amount is positive (adding to totalSavings)
costValue is variable created to convert costString from a String into a Double for the calculations.

```
142     //Function for saving the "Expense" inputted
143     private func saveExpense() {
144         //if "Expense" was selected, the inputted amount would become negative (subtracted from the total savings),
                after first changing costString from a string into a double
145         if selectedCategory == "Expense" {
146             let costValue = (Double(costString) ?? 0.0) * -1
147             let newExp = Expense(name: name, cost: costValue)
148             context.insert(newExp)
149         }
150         else //in this case, if "Income" was selected costString is simply turned from a String into a Double
151         {
152             let costValue = Double(costString) ?? 0.0
153             let newExp = Expense(name: name, cost: costValue)
154             context.insert(newExp)
155         }
156         dismiss()
157
158     }
159 }
```

Figure 13: Use of if statements to add/subtract amounts from totalSavings

Calculator Page

Use of "**Enum**", short for enumeration, to establish a collection of named constants, known as enumerators, each linked with an integer value. This creates the basic numbers and operators needed to form a calculator.

Then, a **switch statement** is used to execute a block of code among many alternatives. Each case represents a different button on the calculator (Figure 14).

```
8  import SwiftUI
9
10 //"Enum" is short for enumeration. It defines a common type for a group of related values and enables developers to establish a collection of named constants,
       known as enumerators, each linked with an integer value.
11 enum CalcButton: String {
12     case one = "1", two = "2", three = "3"
13     case four = "4", five = "5", six = "6"
14     case seven = "7", eight = "8", nine = "9"
15     case zero = "0", add = "+", subtract = "−"
16     case divide = "÷", multiply = "×", equal = "="
17     case clear = "AC"
18
19     //Switch statement is used to assign colors to buttons of the same type at once (e.g. operators, equal sign, single digits (0-9), and the All Clear sign)
20     var buttonColor: Color {
21         switch self {
22         case .add, .subtract, .multiply, .divide, .equal:
23             return Color(red: 0.04, green: 0.43, blue: 0.25)
24         case .clear:
25             return Color(.lightGray)
26         default:
27             return Color(.darkGray)
28         }
29     }
30 }
```

Figure 14: Use of Enum and switch statement for Calculator (iOS Academy)

Constructing the Calculator View: (see in-line comments)

```
37 //Calculator View
38 struct Calculator: View {
39
40     //State allows you to make a variable that can be automatically changed and updated
41     //"0" is the initial/default value displayed on the calculator
42     @State var value = "0"
43
44     //When an operation button is pressed, output value automatically displays a 0, for user to input another number
45     @State var runningNumber = 0
46     //This variable tracks the current mathematical operation
47     @State var currentOperation: Operation = .none
48
49     //Creating a 2-Dimensional array called CalcButton to store all buttons on the calculator
50     let buttons: [[CalcButton]] = [
51         [.add, .subtract, .multiply, .divide],
52         [.seven, .eight, .nine, .zero],
53         [.four, .five, .six, .clear],
54         [.one, .two, .three, .equal]
55     ]
```

Figure 15: Use of @State variables and a 2D array

Creating a reusable function to perform the actual calculations:

```
95      //Function for what to do when a button is tapped/clicked
96      func calculateIt(button: CalcButton){
97          switch button {
98          case .add, .subtract, .multiply, .divide, .equal:
99              //If statement to determine which operator was tapped/clicked
100             if button == .add {
101                 self.currentOperation = .add
102                 self.runningNumber = Int(self.value) ?? 0 //"?? makes it optional"
103             }
104             else if button == .subtract {
105                 self.currentOperation = .subtract
106                 self.runningNumber = Int(self.value) ?? 0
107             }
108             else if button == .multiply {
109                 self.currentOperation = .multiply
110                 self.runningNumber = Int(self.value) ?? 0
111             }
112             else if button == .divide {
113                 self.currentOperation = .divide
114                 self.runningNumber = Int(self.value) ?? 0
115             }
116             else if button == .equal {
117                 let runningValue = runningNumber
118                 let currentValue = Int(self.value) ?? 0
119                 switch currentOperation {
120                 case .add: value = "\(runningValue + currentValue)"
121                 case .subtract: value = "\(runningValue - currentValue)"
122                 case .multiply: value = "\(runningValue * currentValue)"
123                 case .divide: value = "\(runningValue / currentValue)"
124                 case .equal, .none:
125                     break
126                 }
127             }
```

Figure 16: A function for the actual calculations

```
            //If button pressed is not equal, the calculator automatically resets to display a "0" for a new entry (e.g. after an add, subtract, multiply, or
                divide button is tapped, the calculator resets to 0 for a new number to be entered)
            if button != .equal {
                value = "0"
            }
            //This sets the value displayed on the calculator (value) to "0", effectively clearing the current input or calculation.
        case .clear:
            value = "0"
            //If the tapped button does not match the conditions above (not equal to .equal or .clear), this block is executed. It takes the number from the
                button.rawValue. If the current value displayed on the calculator is "0", it replaces it with the number. If the current value is not "0", the
                number is appended to the existing value. This ensures that numbers are concatenated correctly when the user enters multiple digits.
        default:
            let number = button.rawValue
            if self.value == "0"
            {
                value = number
            }
            else
            {
                //Adding the digits
                value += number
            }
```

Figure 17: A continuation of the function for the actual calculations