Criterion C: Development

Table of Contents

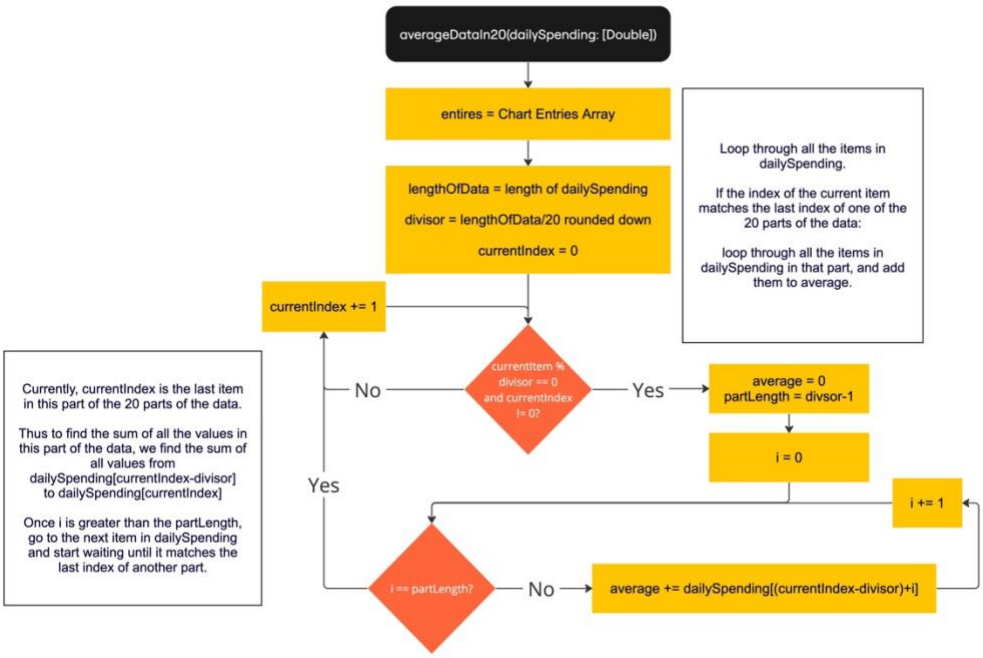
Dependencies

Dependency	Purpose
`UIKit`	Creation of user interface, control of gestures, and control of device resources.
`CoreData`	Database creation and control.
`DropDown`	Creation of drop-down functionality in home screen.
`DGCharts`	Creation and styling of all charts.

Advanced Algorithms/Data manipulation

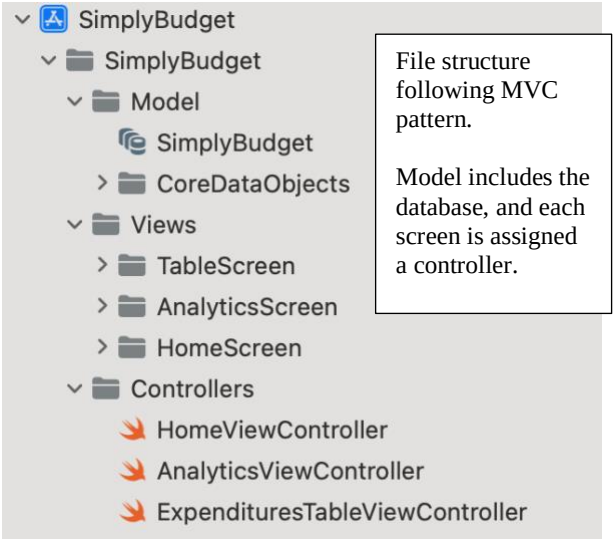
Creating a chart of daily spending over time was a particularly challenging task. At first, it required me to manipulate the data from all expenditures to an array of expenditures made every day and take the sum of these expenditures. Afterwards I found the graph created was incredibly jagged and the DGCharts library only allowed curving of sharp corners, not actually averaging the data to form a smooth curve. Thus, after creating a function that makes the graph, following the flowchart about making a line chart presented earlier, I created a function that splits the data into 20 parts and takes the average of each part shown below. Before this, however, thinking about how to do this task, I realized that while the code should be short, it was more complex than I thought and so I made the flowchart shown below.

```
12  class AnalyticsViewController: UIViewController {
164     func averageDataIn20(for dailySpendingData:[Double]) -> [ChartDataEntry] {
165         // Takes the average of every 20th part of the data and returns these averaged chart entries.
166
167         var entries = [ChartDataEntry()]
168         var counter = 0 // Current index of the dailySpendingData.
169
170         for _ in 0...(dailySpendingData.count - 1) {
171             // Setting average value to be changed later, and average divisor to be one 20th of the data set.
172             var averageValue = 0.0
173             let averageDivisor = Int(dailySpendingData.count / 20)
174
175             // Everytime the current day is at a 20th of the whole data, take the average of all the days within that part of the whole data.
176             if counter % averageDivisor == 0 && counter != 0 && averageDivisor != 0 {
177
178                 for i in 0...averageDivisor-1 {
179                     averageValue += dailySpendingData[counter - averageDivisor + i]
180                 }
181
182                 let finalAverage = averageValue/Double(averageDivisor)
183
184                 // Every 20th part append this entry to data entries (should have 20 in total).
185                 entries.append(ChartDataEntry(x: Double(counter), y: finalAverage))
186             }
187             counter += 1
188         }
189         // Return entries.
190         return entries
191     }
```



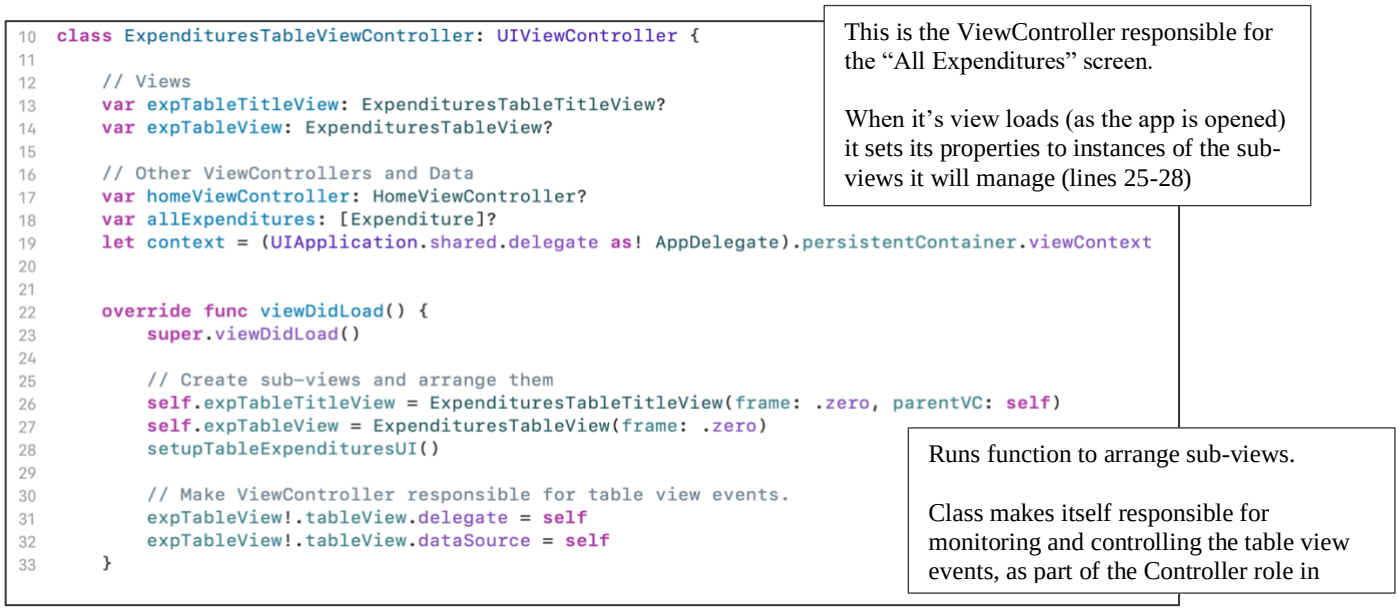
MVC Design Pattern

The **Model-View-Control design pattern** was used to allow for faster development, easier debugging, and a logical organization of files and classes. The MVC requires code that controls the data model, presenting information, and controlling information be separated into different objects (GeeksforGeeks, 2018). In this system the view and the model never interact directly with one another, rather the view-controller is responsible for this role. This was something done throughout my project, and evidence of this for the “All Expenditures” screen is shown in the images below:



File structure following MVC pattern.

Model includes the database, and each screen is assigned a controller.



```swift
class ExpendituresTableViewController: UIViewController {

    // Views
    var expTableTitleView: ExpendituresTableTitleView?
    var expTableView: ExpendituresTableView?

    // Other ViewControllers and Data
    var homeViewController: HomeViewController?
    var allExpenditures: [Expenditure]?
    let context = (UIApplication.shared.delegate as! AppDelegate).persistentContainer.viewContext


    override func viewDidLoad() {
        super.viewDidLoad()

        // Create sub-views and arrange them
        self.expTableTitleView = ExpendituresTableTitleView(frame: .zero, parentVC: self)
        self.expTableView = ExpendituresTableView(frame: .zero)
        setupTableExpendituresUI()

        // Make ViewController responsible for table view events.
        expTableView!.tableView.delegate = self
        expTableView!.tableView.dataSource = self
    }
```

This is the ViewController responsible for the "All Expenditures" screen.

When it's view loads (as the app is opened) it sets its properties to instances of the sub-views it will manage (lines 25-28)

Runs function to arrange sub-views.

Class makes itself responsible for monitoring and controlling the table view events, as part of the Controller role in

As shown, the "All Expenditures" controller creates instances of its sub-views and manages all types of interactions with the database (such as creating the table data, shown below).

```swift
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // Pick a custom cell (defined in Views) for UI, pass in data.
    do {
        // Fetch the all the expenditures from the user in the database and sort them in chronological order.
        let appUser = try context.fetch(User.fetchRequest())[0]
        self.allExpenditures = (appUser.totalSpending?.allObjects as! [Expenditure]).sorted(by: {
            $0.timeStamp!.compare($1.timeStamp!) == .orderedDescending
        })

        // Get cell UI design from ExpendituresTableCell (View)
        guard let cell = tableView.dequeueReusableCell(withIdentifier: ExpendituresTableCell.identifier, for: indexPath) as?
            ExpendituresTableCell else {
            fatalError("The table view could not dequeue a ExpendituresTableCell.")
        }

        // Configure data in dequeued cell.
        let cellCategory = allExpenditures![indexPath.row].category
        let cellAmount = allExpenditures![indexPath.row].amount
        let cellTime = allExpenditures![indexPath.row].timeStamp

        cell.configure(category: cellCategory!, amount: cellAmount, timeStamp: cellTime!)
        return cell
    }
    catch {
        fatalError("Error fetching budget data for table view.")
    }
}
```

This is a delegate function (explained later) responsible for determining the data in every cell in the table view. Accordingly, since the **Controller** handles all interaction with the database, in this class fetches all the expenditures from the database (lines 84-85) and updates the cells accordingly (lines 93-99). To make this code more readable I use Swift's **short-hand argument**

names ($0, $1) within the inline closure to order the data fetched in chronological order (line 85) (Apple Inc., 2023a). This pattern allows classes to solely focus on what their roles are which prevented tons of redundant code, as otherwise I would have had to create methods for fetching data in each of the numerous views, and made the development process much faster.

Delegate Pattern

In Object Oriented Programing, the **Delegate pattern** is when an object utilizes another object (or delegate) to carry out tasks (Apple Inc., 2023b). This was precisely what was required when making the "All Expenditures" screen as the ViewController would act as the delegate of the UITableView element.

```
68  extension ExpendituresTableViewController: UITableViewDelegate, UITableViewDataSource {
69      func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
70          // How many table rows we want.
71          do {
72              let appUser = try context.fetch(User.fetchRequest())[0]
73              return appUser.totalSpending?.count ?? 0
74          }
75          catch {
76              fatalError("Error fetching budget data for table view.")
77          }
78      }
79
80      func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
81          // Pick a custom cell (defined in Views) for UI, pass in data.
82          do {
83              // Fetch the all the expenditures from the user in the database and sort them in chronological order.
84              let appUser = try context.fetch(User.fetchRequest())[0]
85              self.allExpenditures = (appUser.totalSpending?.allObjects as! [Expenditure]).sorted(by: {
                    $0.timeStamp!.compare($1.timeStamp!) == .orderedDescending
86              })
87
88              // Get cell UI design from ExpendituresTableCell (View)
89              guard let cell = tableView.dequeueReusableCell(withIdentifier: ExpendituresTableCell.identifier, for: indexPath) as?
                    ExpendituresTableCell else {
90                  fatalError("The table view could not dequeue a ExpendituresTableCell.")
91              }
```

As a delegate, one of the functions required is one that returns how many rows you will have in your table.

The other function required is one that returns a cell for every row

Extending my ViewController class I **inherit the class** of UITableViewDelegate and are required to include the functions shown above. It is impossible for the TableView to know how many rows or what data it should have in the table, which makes the delegate pattern work perfectly in this scenario, allowing my ViewController which has access to the data to communicate this information to the TableView without having to create redundant methods inside the TableView class itself. Thus, I used this pattern to successfully facilitate the

communication of data, prevent redundant code and to maintain consistency in the MVC design pattern.

Error Handling

Throughout previously shown code, I effectively handle possible errors in the program. Primarily, errors from a failure to successfully fetch data from the database, which could be the result from a malfunction in the client's device. In these scenarios rather than keep the program running and ignore errors, which could lead to a failure to save data and more errors, I call a fatalError which terminates the application. These were instrumental in the development process as it ensured not only that I considered events that could cause more errors had the error not been raised but also provided information about the error in question. This greatly facilitated debugging and ensured that the client does not proceed with a faulty application, and that the data in the database remains consistent and not incomplete.

```
10  class HomeAddExpenditureView: UIView {
32      //MARK: - (Add Expenditure) Functions
33      @objc func addExpenditureButtonClicked() {
34          // When plus button is clicked on any one of the categories. Add expenditure to database.
35
36          guard let textFieldText = addExpenditureTextField.text else { return }
37
38          if textFieldText != "" {
39              // Add new expenditure to database
40              do {
41                  let appUser = try context.fetch(User.fetchRequest())[0]
42                  let newExpenditure = Expenditure(context: context)
43
44                  newExpenditure.timeStamp = Date()
45                  newExpenditure.amount = Int64(textFieldText)! // Only numbers allowed in textfield.
46                  newExpenditure.category = self.currentCategoryName! // Category name is set as soon as button is clicked.
47                  appUser.addToTotalSpending(newExpenditure)
48                  self.addExpenditureTextField.text = ""
49                  self.endEditing(true)
50
51                  do {
52                      // Try to add expenditure to the database.
53                      try context.save()
54                  }
55                  catch {
56                      // If adding expenditure fails abort operations.
57                      fatalError("Error: failed to save data.")
58                  }
59
60              }
61              catch {
62                  // If you cannot fetch data from the database abort operations.
63                  fatalError("Error fetching budget data")
64              }
```

Gesture Recognition

To improve the client's experience, the app presents the "All Expenditures" and "Analytics" screens through swipe gestures. This was combined with the default animated presentation of presenting a view controller from the right for the "Analytics" screen, however a custom animation had to be made for the "All Expenditure" screen as it should be presented from the left. This was cleverly done in the form of a class extension (discussed below) and in combination with the other animation allowed for a smooth user experience.

```
134     //MARK: - Swipe Gestures
135     func setupSwipeLeftGesture() {
136         // Create gesture recognizer.
137         let swipeLeftGesture = UISwipeGestureRecognizer(target: self, action: #selector(didSwipeLeft))
138         swipeLeftGesture.direction = .left
139 
140         // Add gesture recognizer to view.
141         self.view.addGestureRecognizer(swipeLeftGesture)
142     }
143 
144     @objc func didSwipeLeft() {
145         // Function called whenever the user swipes left.
146         let analyticsVC = AnalyticsViewController()
147         self.navigationController?.pushViewController(analyticsVC, animated: true)
148     }
149 
150     func setupSwipeRightGesture() {
151         // Create gesture recognizer.
152         let swipeRightGesture = UISwipeGestureRecognizer(target: self, action: #selector(didSwipeRight))
153         swipeRightGesture.direction = .right
154         // Add gesture recognizer to view.
155         self.view.addGestureRecognizer(swipeRightGesture)
156     }
157 
158     @objc func didSwipeRight() {
159         // Function called whenever the user swipes right.
160         self.navigationController?.pushViewControllerFromLeft(controller: expendituresTableVC!)
161     }
```

Programmatic UI through Swift Closures

To allow for precise and responsive UI, UIKit view elements were arranged **programmatically** (rather than using the visual feature on XCode), this greatly facilitates customizability and through constraints, ensures that even on UI will still look great even on larger or smaller iPhone

screens. Furthermore, Swift **closures** were used to for the creation of UI objects to make code more readable and logical, shown in the image below when creating the Edit Budget Menu View. Thus, I created a reactive and easily customizable UI while maintaining readable code through programmatically arranging UIKit view elements.

```swift
class HomeBudgetMenuView: UIView {

    var darkBackgroundLayer: UIView = {
        let view = UIView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = UIColor(red: 0, green: 0, blue: 0, alpha: 0.4)
        return view
    }()

    var budgetMenuBackground: UIView = {
        let view = UIView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.layer.cornerRadius = 5
        view.backgroundColor = UIColor(red: 0.95, green: 0.99, blue: 0.96, alpha: 1.00)
        return view
    }()

    var budgetMenuTextField: UITextField = {
        let textField = UITextField()
        let centeredParagraphStyle = NSMutableParagraphStyle()
        textField.translatesAutoresizingMaskIntoConstraints = false
        textField.layer.cornerRadius = 10
        textField.clipsToBounds = true
        textField.backgroundColor = .white
        textField.placeholder = "Enter Quantity..."
        textField.setLeftPaddingPoints(20)
        textField.keyboardType = .numberPad
        return textField
    }()
```

Creation of a TextField object with styling using a Swift closure.

The closure allows for an object that is created and has its properties edited in multiple lines, be returned, and set as the

```swift
class HomeViewController: UIViewController {
    private func setupHomeUI() {
        // Sets-up UI for Home Screen.

        // Add the title or TimeRangeView at the top of the screen.
        view.addSubview(timeRangeView!)
        timeRangeView!.topAnchor.constraint(equalTo: view.topAnchor, constant: 0).isActive = true
        timeRangeView!.heightAnchor.constraint(equalToConstant: 150).isActive = true
        timeRangeView!.widthAnchor.constraint(equalTo: view.widthAnchor, multiplier: 1).isActive = true

        // Add the number displaying the budget remaining for time-range selected in the middle of the screen.
        view.addSubview(budgetCounterView!)
        budgetCounterView!.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        budgetCounterView!.centerYAnchor.constraint(equalTo: view.centerYAnchor, constant: -135).isActive = true
        budgetCounterView!.heightAnchor.constraint(equalToConstant: 200).isActive = true
        budgetCounterView!.widthAnchor.constraint(equalTo: view.widthAnchor).isActive = true

        // Add all the category buttons at the bottom of the view.
        view.addSubview(categoriesView!)
        categoriesView!.bottomAnchor.constraint(equalTo: view.bottomAnchor).isActive = true
        categoriesView!.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        categoriesView!.heightAnchor.constraint(equalToConstant: 400).isActive = true
        categoriesView!.widthAnchor.constraint(equalTo: view.widthAnchor).isActive = true

        // Add the categories add menu to the whole screen (it is hidden by default)
        view.addSubview(categoriesView!.addMenu!)
        categoriesView!.addMenu!.center = view.center
        categoriesView!.addMenu!.widthAnchor.constraint(equalTo: view.widthAnchor, multiplier: 1).isActive = true
        categoriesView!.addMenu!.heightAnchor.constraint(equalTo: view.heightAnchor, multiplier: 1).isActive = true

        // Add the edit budget menu to the whole screen (it is hidden by default)
        view.addSubview(categoriesView!.budgetMenu!)
        categoriesView!.budgetMenu!.center = view.center
        categoriesView!.budgetMenu!.widthAnchor.constraint(equalTo: view.widthAnchor, multiplier: 1).isActive = true
        categoriesView!.budgetMenu!.heightAnchor.constraint(equalTo: view.heightAnchor, multiplier: 1).isActive = true
    }
```

Function that **programmatically** arranges all the sub-views in the Home Screen, executed by the Home Screen View Controller.

Class Extensions

There were numerous occasions where a method would be used to modify a data type or an object but that the object itself did not have defined. For example, adding commas to a number would be a greatly useful method for Int objects, but being unable to modify the source code of the Int class it would require many of the same functions or a new class entirely. Thus, to reduce redundancy and improve code readability I made use of Swift extensions which add additional functionality to an already existing class (Apple Inc., 2023c). This was a very effective way to add functionality required for the application to the most logical class.

```swift
extension Int {
    func addCommas() -> String {
        let numberFormatter = NumberFormatter()
        numberFormatter.numberStyle = .decimal
        return numberFormatter.string(from: NSNumber(value: self))!
    }
}

extension Calendar {
    private var currentDate: Date { return Date() }

    func isDateInThisMonth(_ date: Date) -> Bool {
        return isDate(date, equalTo: currentDate, toGranularity: .month)
    }

    func isDateInThisYear(_ date: Date) -> Bool {
        return isDate(date, equalTo: currentDate, toGranularity: .year)
    }
}

extension UINavigationController {
    func pushViewControllerFromLeft(controller: UIViewController) {
        let transition = CATransition()
        transition.duration = 0.3
        transition.type = CATransitionType.push
        transition.subtype = CATransitionSubtype.fromLeft
        transition.timingFunction = CAMediaTimingFunction(name: CAMediaTimingFunctionName.easeInEaseOut)
        view.window!.layer.add(transition, forKey: kCATransition)
        pushViewController(controller, animated: false)
    }
```

Extension to Int class returning a string of the number with commas.

Extension to the Calendar class allowing for a quicker way of checking whether a given date is in the current year or month.

Extension to the UINavigationController class allowing a ViewController to be presented from the left.

Word Count: 1118

Reference List

Apple Inc. (2023a). *Documentation*. [online] docs.swift.org. Available at: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/closures/.

Apple Inc. (2023b). *Documentation*. [online] docs.swift.org. Available at: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/protocols/.

Apple Inc. (2023c). *Documentation*. [online] docs.swift.org. Available at: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/extensions/.

GeeksforGeeks (2018). *MVC Design Pattern - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/mvc-design-pattern/.

Sarkar, P. (2022). *Delegation Pattern: An effective way of replacing Android's Base Activity with native Kotlin support*. [online] Medium. Available at: https://prokash-sarkar.medium.com/delegation-pattern-an-effective-way-of-replacing-androids-baseactivity-with-native-kotlin-support-b00dee007d69 [Accessed 17 Oct. 2023].