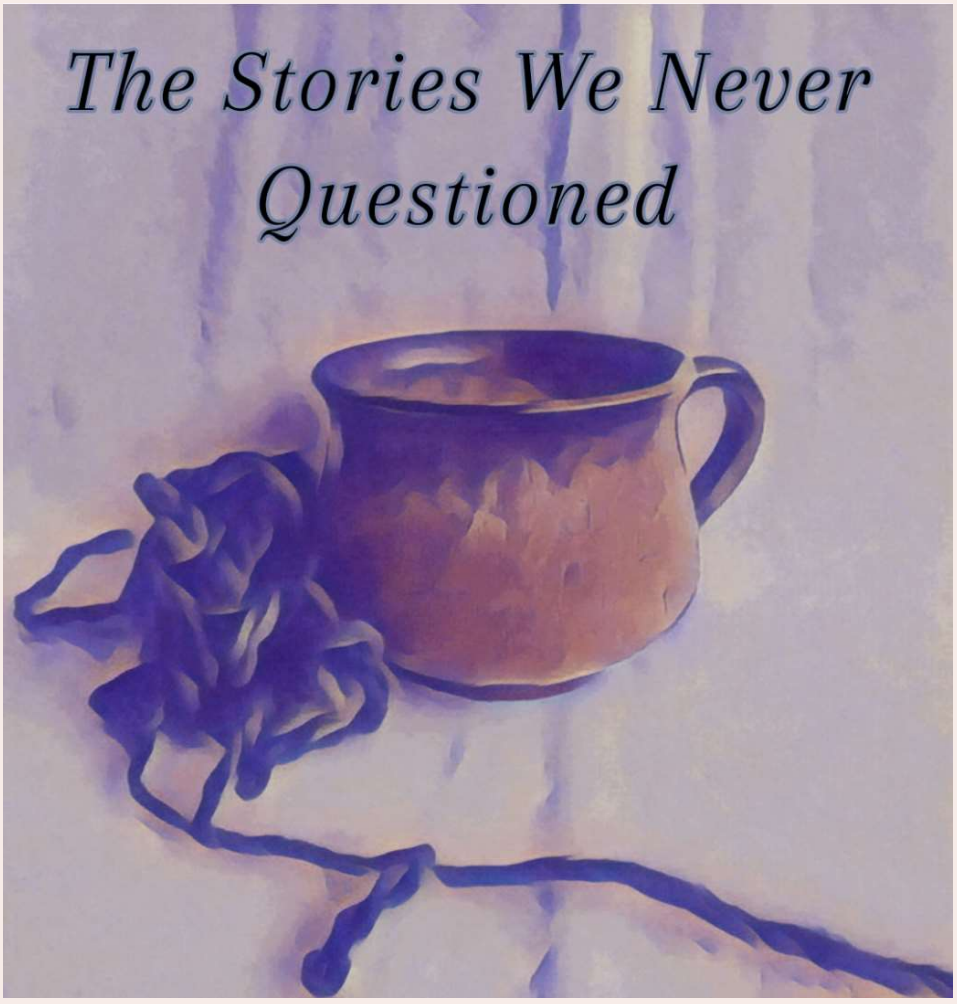
12 / 08 / 2020
The Giving Child
Field Session Team Fall 2020



Andrew Faalevao,
Kenny Lieu,
Emelyn Pak,
Dane Pham,
Hannah Taylor

TABLE OF CONTENTS

INTRODUCTION

The Giving Child is a 501(c)(3) non-profit organization focused on empowering children to be the best global citizens they can be. After the Black Lives Matter protests over the summer, The Giving Child recognized a need for addressing systemic racism in our communities. This application in particular is The Giving Child's effort to combat implicit racial bias in our older neighbors and friends. After spending most of their existence helping children grow in a healthy way, The Giving Child decided to pivot to an overlooked segment of the population. Even though seniors are older doesn't mean they are done learning. What started as a need to educate the founder's parents on systemic racism against African Americans turned into a full-blown effort to educate the masses of senior citizens against the implicit biases they have held and nurtured for a majority of their lives.

This app is the first app The Giving Child is making that's not a game. Instead, since the intended audience skews older, The Giving Child wanted an intuitive, easy-to-use application that directly links to their web resources about racial bias. On their website, The Giving Child has resources like stories of racism, tea-time, and lists of local minority-run businesses on their website. The application itself serves as an easy way for seniors to access these resources. Our team created an intuitive, easy-to-use app that makes it easy for users to navigate between the different pages of The Giving Child website.

High-level Product Vision

Vision: *"Let's go on this journey of internal bias together with the baby boomer generation."*

Mission: *Create a simplistic app that guides members of older generations on lessons in systemic racism and bias.*

High-Level Description: *The clients want to create an application that urges older Americans to look inwards concerning the ongoing racial unrest in the country.*

The app will have an onboarding tutorial for the new users. Every day, there will be a notification for journaling a word that describes their feelings for each day. Additionally, if they are comfortable learning that day, they will be redirected to the client's website for the daily lesson. There will also be pages dedicated to experts, the community, about us, developers, and a feelings tracker.

REQUIREMENTS

Functional Requirements

This app must allow users to navigate through the different pages that are provided, track their progress, and access daily lessons.

First-Time User:

An introduction and tutorial to the app is shown for the users to click through. A couple pop-ups will also appear. One will ask the user to allow notifications for the app, and the other will display a disclaimer informing the user that the app cannot cure racism.

Finally, once the disclaimer is acknowledged, there is a tutorial with a short introduction to the content of the app that the users can click through to learn how to use the app to best motivate their learning experience. After completing the tutorial, the app will open to the home page.

Returning User:

For returning users opening the app on their own or through following the daily notification, the app opens to the home page which has a few buttons that lead to various pages in the app. The first links to the menu page which holds links to all of the pages in the app. The next allows the user to go straight to the Feelings Tracker to log their feelings for the day before beginning their lesson. The final button opens an introduction video to the app on YouTube from our clients.

The menu page of the app holds buttons that link to the pages of the app:

Experts Page:
- *Displays a link to the "Experts" section of The Giving Child website*

Community Page:
- *Displays a link to the "Community" section of The Giving Child website*

Stories We Were Told Page:
- *Displays a link to "The Stories" section of The Giving Child website*

Feelings Tracker Page:
- *Allows user to view all their logged feelings as well as the date they were logged on*
- *Allows user to add a new feeling to the log*

Team Credits Page:
- *Displays a link to the "Team Credits" section of The Giving Child website*

About Us Page:
- *A page that displays a link to the "About Us" section of The Giving Child website*

Time to Learn:
- *Displays an option for the user to either complete their lesson for the day or take a break*
- *Links to "The Stories" section of The Giving Child website*
- *Links to the "Meditations" section of The Giving Child website*

Library Sketch Page (stretch goal):
- *A physical representation of the user's progress*
- *A sketchbook like drawing will be displayed in black and white and each lesson completed will color in a portion of the drawing*

The notification received by the user each day will remind the user to visit the app to log their feelings and complete their daily lesson.

Non-Functional Requirements

iOS development:

The app must satisfy the guidelines laid out by Apple to list on the app store.

The app must be written in Swift which is the language compatible with iOS app development.

The app must be able to integrate with the content and website provided by The Giving Child

Application Architecture

Because iOS applications require a Mac for development, our team utilized MacinCloud, a service providing cloud Mac servers, to gain access to the proper tools necessary for the development of our application. Once we gained access to the Mac server, our team utilized XCode, Apple's IDE for iOS development, to create our application. XCode utilizes the Swift programming language for the design of iOS applications and software.

Technical Design Issues

Since our access to iOS development tools was through a remote server, the main technical issue that we faced as a team was dealing with server connection issues with MacinCloud throughout the semester. MacinCloud takes a bit of time to initialize and start the session, and there is visible lag as the mouse is moved. Along with this, our team consistently experienced issues with logging in to our respective MacinCloud servers. This issue proved to be a major hindrance to our work timeline as the only way to solve this issue was either by contacting a MacinCloud representative or waiting until the server was back online. Despite these issues, our team was still able to utilize MacinCloud to complete the development of our application.

Design Diagrams

The following diagrams illustrate and describe the functionality of our app and the interactions between the different aspects. Figure 1 below is the flowchart that shows the functionality of the application and the interactions between the different pages. A more readable image of the flowchart can be seen in Appendix A1.

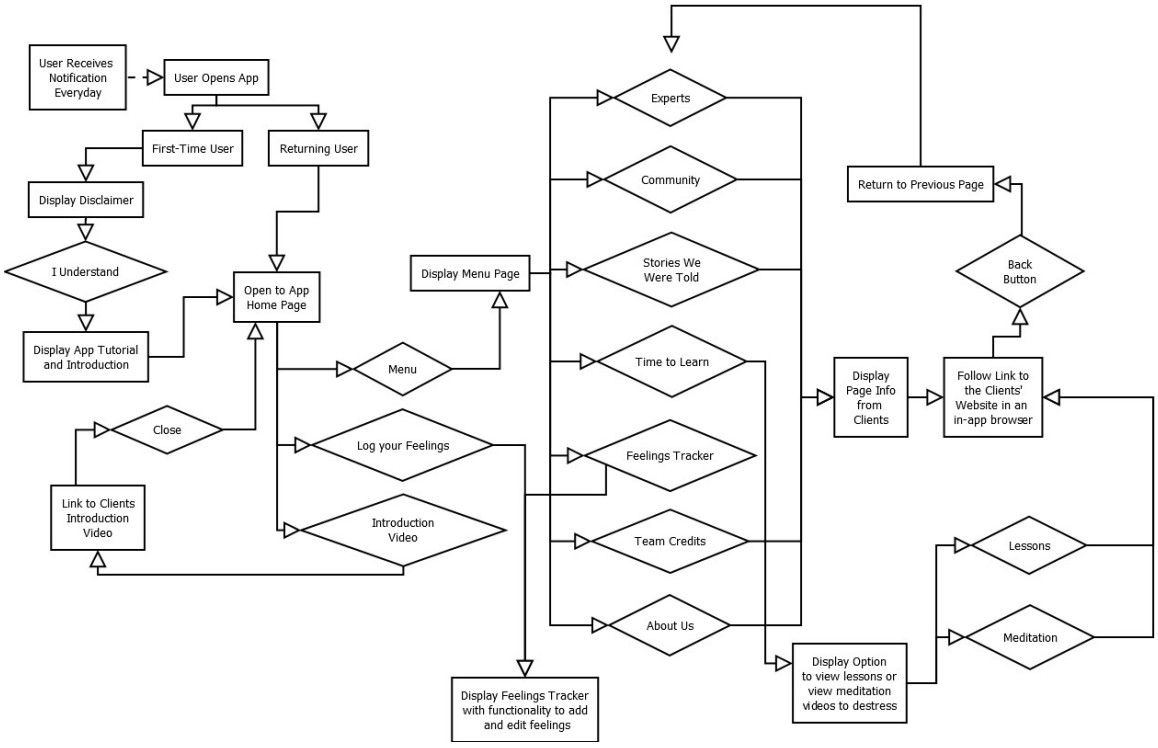


Figure 1: App Flowchart

The app's pages can also be visualized through the storyboards in Figure 2 provided by our clients. They illustrate the visual style of the app that The Giving Child wanted the app to look like. Close up images of the storyboards can be seen in Appendix A2.

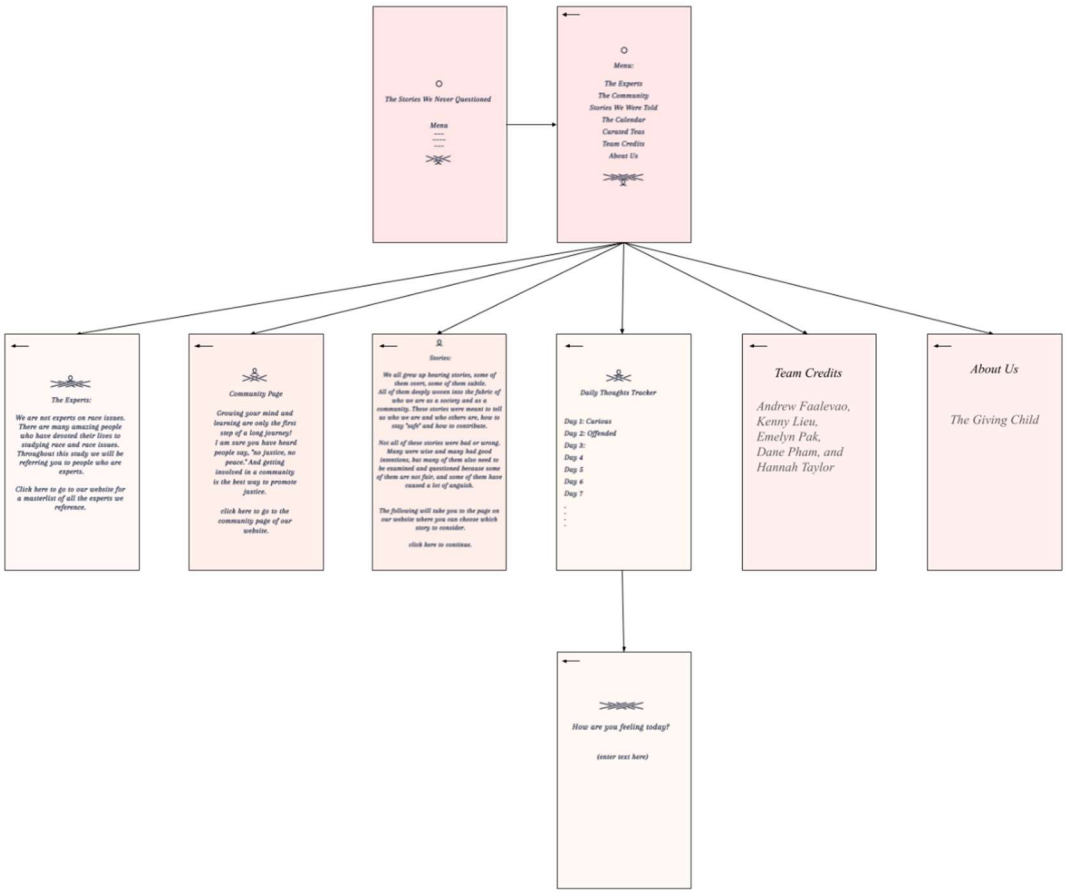


Figure 2: Storyboards

Each page within the app inherits from the Navigation Controller class. In the below diagram, there are 3 child classes shown (Menu Scene, The Experts Scene, and Meditation Web Scene). Each child inherits the properties needed to successfully navigate to the previously accessed page or other pages within the app, when possible. As an example, the Menu Scene page that is displayed can navigate to The Experts Scene page and vice versa. The Meditation Web Scene can navigate to the Time to Learn Scene, which is not shown in the diagram in Figure 3. The inherited function serves to construct the unique navigation on each page. Every page within the app is a child class of the Navigation Controller, however, only 3 of these are shown in order to reduce repetition. A readable version of the UML diagram below can be seen in Appendix A3.

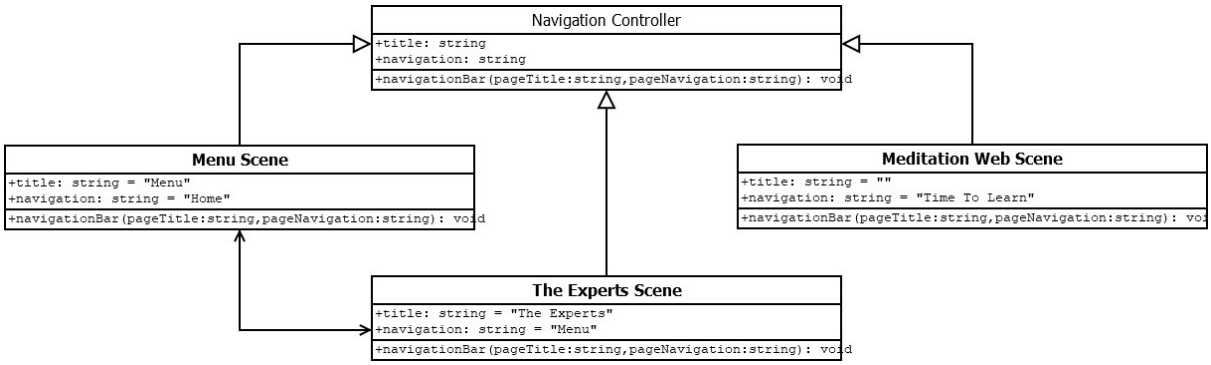


Figure 3: Navigation UML Diagram

Each individual log is stored in an array in LoggerStorage. (See Appendix A7 for LoggerStorage functions/implementation) The individual logs are accessed by a unique logIndex and are then pushed to the UITableViewCell class for viewing. The Log class stores a unique ID, the user's feeling, and the time which they added/edited the feeling. The four empty classes on the top of Figure 4 below represent the flow of the UI for the Feelings Logger. There is a ChangeViewController which controls what view is shown to the user. The three subviews underneath it represent the table view where the user can see all their feelings logged, a detail view where they can click and see what they wrote, and an individual cell where the user can edit their feelings. The only way the user can access the individual cell is by clicking the "Edit" button on the detail view or the "+" button when trying to create a new feeling to log. The diagram below can also be seen in Appendix A4.

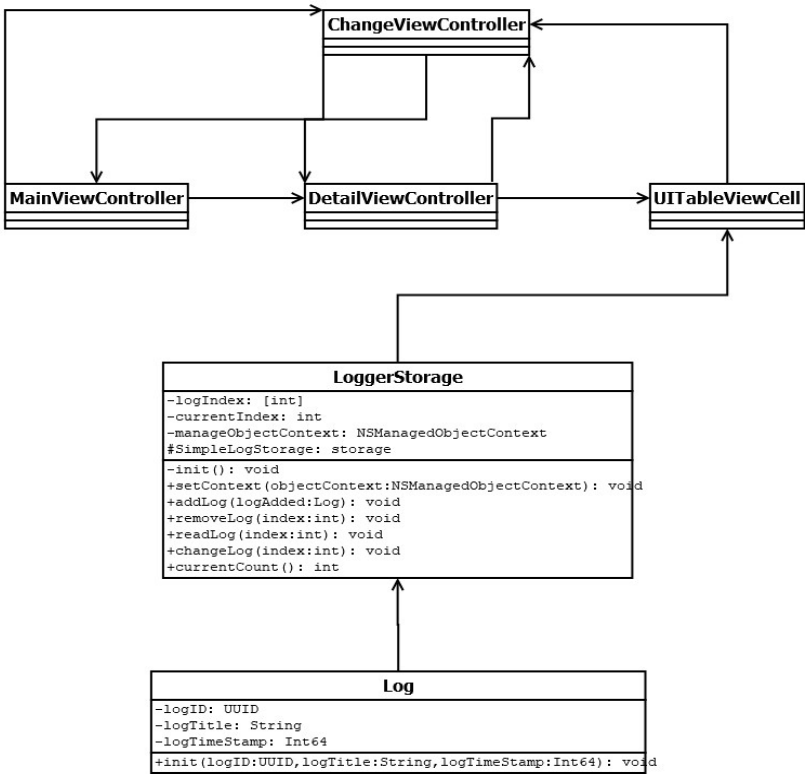


Figure 4: Feelings Tracker UML Diagram

Dynamic Font Sizing

Since a large part of our motivation for choosing how to implement the app was centered around accessibility, it was important that our app be optimized for all different Apple devices and screen sizes. In order to dynamically size the text, we created two classes, one for dynamically sizing labels and one for dynamically sizing buttons.

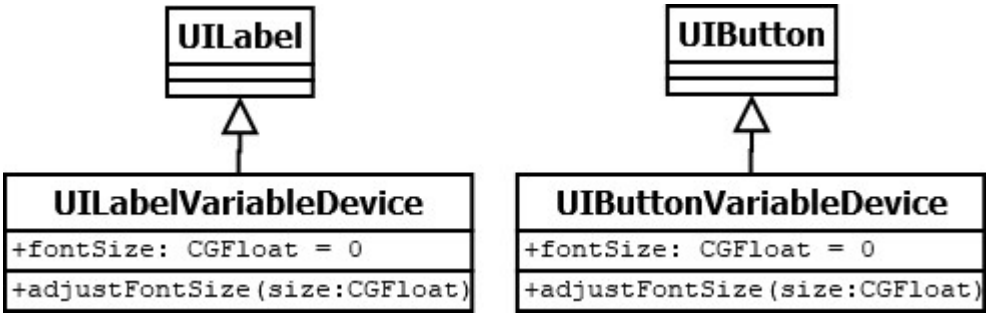


Figure 5: Dynamic Font Sizing UML Diagram

Figure 5, the above diagram, illustrates the classes created to facilitate the adjustment of font sizes. Due to the differences between UILabels and UIButtons, the creation of two classes with almost identical implementations was necessary. UILabel and UIButton store their font information differently, so the classes differ only in how the changes are stored when adjusting them to the screen size. Beyond this, the theory behind the implementation is the same. Within *adjustFontSize*() the current font name is stored so that the updated font stays the same, a new font object is created to store the updated font and font size before storing it in the appropriate label or button object. Additionally, the current screen width is found from the bounds of the UIScreen and stored to be a part of a switch statement which is a simple way to differ the functionality of adjusting the font size when the width of the screen is different.

Once the width of the screen is known, the switch statement matches with each of the known widths of iPhones, iPods, and iPads. With our implementation, we chose the default font size to match the size of the iPhone 12. The other switch cases either size the label or button down if the width of the screen is smaller or up if the width of the screen is larger than our default in order to keep the text readable on all devices. This new font size is then stored with the original font name in the label or button object being altered. The implementations of these classes can be seen in Appendices A5 and A6.

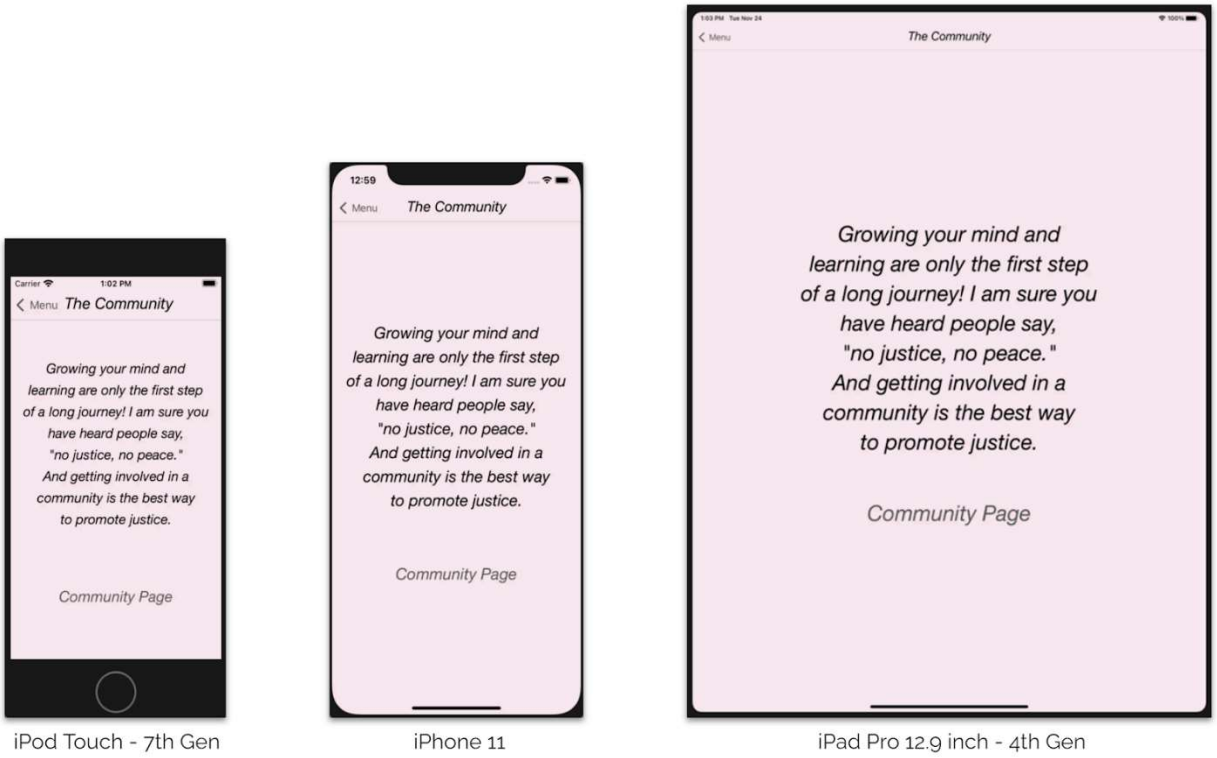


Figure 6: Dynamic Font Sizing at work on 3 different iOS devices

Figure 6 illustrates these classes at work, ensuring the readability of the text on our app. The adjustment of font sizes in our application was included mainly to make sure that our app is accessible on all devices. With the existing functionality available in XCode, the text was either cut off on smaller devices and/or too small to read on larger devices. Due to these limitations, the creation of these classes was necessary.

Feelings Tracker

A key feature our clients wanted implemented was a logger for users to track their daily feelings. This was by far the most technical aspect of our project. To save the data, we use the Core Data Framework to cache data locally on a single device. An equivalent structure to Core Data would be something like a SQL database. Within the Core Data model, we created a single entity called note where multiple attributes are defined such as date and feelings. To track each unique Core Data entity, we assigned it a random ID number and added it to an array of notes. We use the UITextView and UITableView classes to display the data stored in the Core Data model. The first view the user sees, seen in Figure 7, is a table view with all the feelings the user has saved.

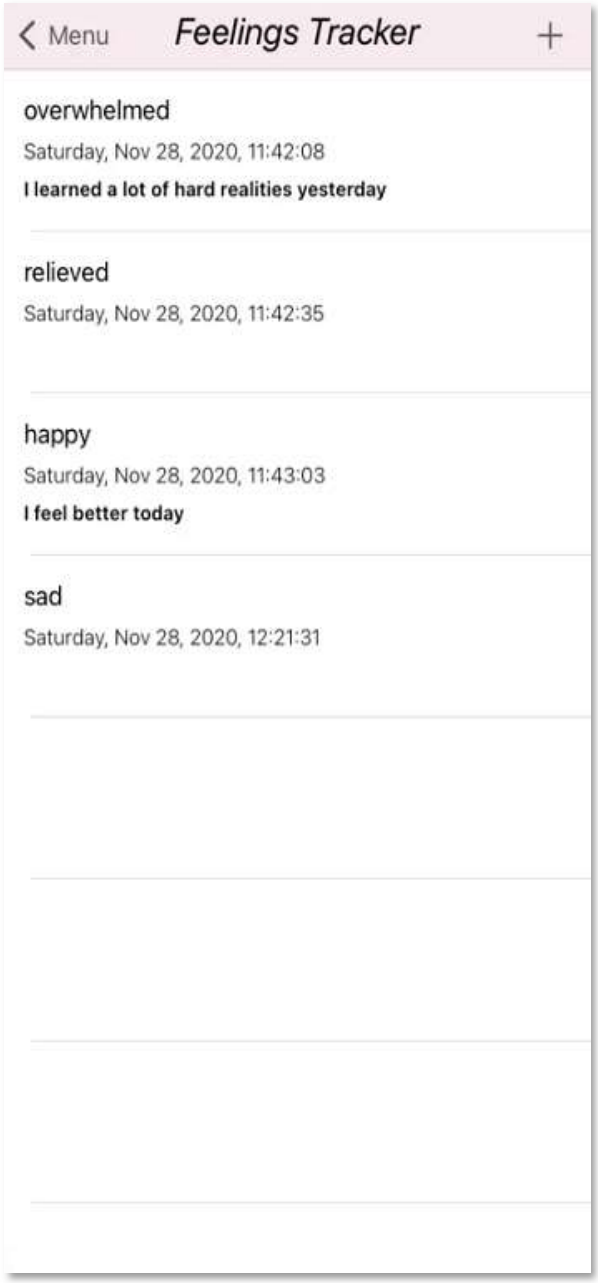


Figure 7: Feelings Tracker Table View

To add a feeling the user can press the "+" button in the top right. To edit a note the user just has to click the note they want to edit. There is a back button in the top left to return the user to the main menu. When creating a feeling, the user can add their feelings and elaborate on them. This view can be seen in Figure 8 below. The date and time are automatically filled in, so the user does not have to manually add them. When editing a feeling, the user just needs to edit the text and click the OK button on the bottom. The time and feeling will automatically update accordingly. If the user wants to go back to the main menu, they just need to click the back button on the logger page, and the app seamlessly switches back. The main portion of the implementation of the Feelings Tracker can be seen in Appendix A7.

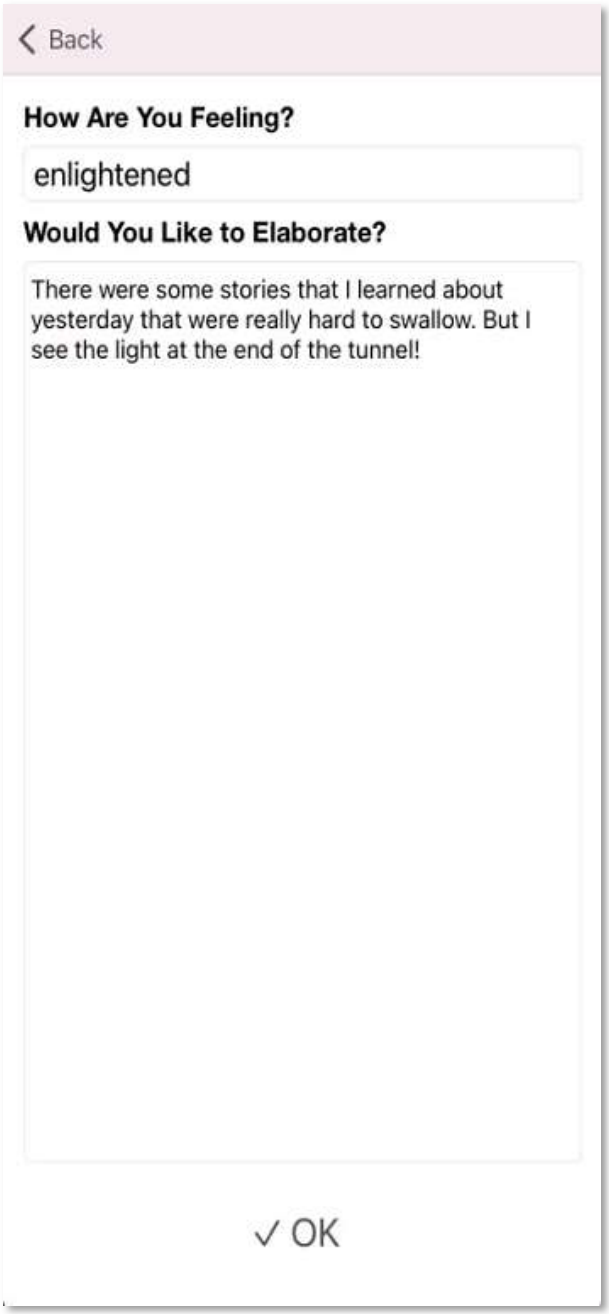


Figure 8: Feelings Tracker Editing View

User Interface Testing

The process of testing the functionality of the user interface to make sure that it is working according to our decided upon specifications.

- Page Navigation
  - Ensured that the buttons on the menu page take the user to the corresponding page on the app
  - Made certain the back button on each page successfully navigated back to the previous page in the app
- Notification Readability
  - Kept the notification wording concise and to the point so that it wasn't so wordy that the user was deterred from following through to the app
- Feelings Tracker Accessibility and Content
  - Tested the navigation between the different pages within the Feelings Tracker
  - Implemented a back button to the Home page if the user accessed the Feelings Tracker from the button on the Home page
  - Implemented a back button to the Menu page if the user accessed the Feelings Tracker from the button on the Menu page
  - Ensured input was logged and displayed correctly
- Link Checking
  - Ensured that when a link is clicked, the application redirects the user to the appropriate webpage
- Navigation Checking
  - Ensured the proper button in the home menu brings up the associated page
  - Made sure there are no "dead-ends" in our application navigation
- Dynamic Font Sizing
  - Used different devices on the built-in emulator to check for automatic font-size adjustments
  - Ensured the application scaled appropriately depending on the screen size

1. Manual Testing: manually test the user interface using the simulator tools built into XCode
   a. The iOS device simulator built into XCode was used to navigate through the app, checking the functionality, mainly of the page navigation and Feelings Tracker page content
   b. Notification readability was tested by editing the content of the notification to be as concise as possible and then ensuring it displayed correctly when the notification was sent

Using this type of testing was not too tedious with our application since its functionality is relatively simple. Since the tools are built right into XCode, it allowed us to continually test the user interface as changes were made. This allowed us to make sure the changes we made did not break the current user interface.

Integration Testing

The process of testing the combined functionality of separate components of software.

- Notification:
  - Create the notification
  - Set a time for the notification to be sent
  - Send calls to the application to open and run when pressed
- Feelings Tracker:
  - Save input globally
  - Display input on feelings tracker page
  - Display input on the feelings tracker page

1. Bottom-Up Testing: testing lower-level components of a process first and using these components to test higher-level functionalities.
   a. Notification: We implemented each step (listed above) before implementing the next (i.e. create the notification before testing the trigger)
   b. Feelings Logger: The same concept applies here. We tested the text box parameters (using user interface testing) before running tests on saving input globally. Tests were run to ensure that the input was saved correctly before trying to display it on the Feelings Tracker page, etc.

Using this type of testing ensured that each lower-level functionality worked correctly, thus creating a solid foundation for more complex functionalities. Testing in this way allowed us to focus on a single component at a time to make sure that the respective component was error-free before building on top of it.

Code Reviews

Code review is a quality assurance activity in which software engineers examine other engineers' code either after the code is implemented or while it is in the process of implementation.

- Weekly Code Reviews
  - Our team conducted weekly code reviews on newly implemented aspects of our application to ensure that the functionality/design was approved by the entire team.
  - Through these meetings, our team was able to ensure that all new features were functional both by themselves and when implemented into the application.

User Acceptance Testing

The process of testing a product amongst a group of targeted users to ensure the quality of software.

- The initial release of our application was tested amongst a small group of older Americans and individuals, including our friends and family members with visual impairments, to test the readability and accessibility of the application.
- From this test, we received feedback that allowed us to gather valuable information
- These tests will allow the team to ensure that our application is easily navigable, making certain that the lessons present on the app are heard by our targeted audience.

Static/Dynamic Program Analysis

The process of analyzing our product during development (Static Program Analysis) and during runtime (Dynamic Program Analysis).

- Static Program Analysis
  - *SwiftLint* is the software that was utilized by our team to conduct Static Program Analysis.
  - *SwiftLint* is a free and open-source XCode plugin that allowed us to set coding style rules that we agreed on as a team.
  - *SwiftLint* enforced these coding style rules by displaying warning/error messages if the rules were broken during development.
  - Consistent coding styles across the board allowed for future changes/updates to be implemented in a much easier manner.
  - Some examples of the coding conventions we used can be seen in Appendix A8.
- Dynamic Program Analysis
  - XCode *Instruments* are the tools that were used for Dynamic Program Analysis
  - *Instruments* are a series of tools built-in to XCode that allowed us to check our code for performance issues, memory issues, reference cycles, and other issues that could have occurred at runtime.
  - *Instruments* helped us to locate potential issues that could have "broken" our application, causing it to fail to compile.

Summary of Testing

We tested our project with a variety of methods as explained above in our Quality Assurance plan to ensure accessibility and functionality. We completed as many of our client's functional requirements as possible within the time frame, and through discussions with our team and our clients deemed some features as future work to be done for the app in the future. Notifications, feelings tracking, and all other functionalities work as expected. We had run into issues regarding editing the feelings once it is saved to the application, but it is now implemented and working.

We kept navigation simple and linear, stemming from the menu page where each path is clearly listed. A majority of the pages contain a link that is displayed clearly and concisely in order to deviate from confusion. We successfully integrated these links as in-app browsers linking to the corresponding pages of our clients' website. Dynamic font sizing is also a key feature that we focused on, given that our target audience is likely to require a larger font size in order to take advantage of this app's full benefits. This addition allows the readability of the app to be maintained regardless of the device it is being accessed from.

Testing Results

Navigation:

The navigation between pages of our app as well as the navigation to the correct page on our clients' website was tested manually using the iPhone and iPad simulators built into XCode. Testing on all of the device simulators manually, we navigated through each link on our home page, menu page, and any pages with in-app browsers and verified that they navigated to the correct/expected location.

Dynamic Font Sizing:

Unit Testing could have been useful for this aspect of the application, but due to the fairly minimal scope of our application, we manually tested this with the XCode iPhone and iPad simulators. We implemented two class mentioned in the technical design section of this report that match the width of the different screen sizes to a size factor that can be applied to the chosen font and font size. Manually testing these classes of labels and buttons, the text they were applied to correctly size up or down on all of the available simulators of various iPhone and iPad screen sizes.

Notification:

The daily notifications were tested by temporarily setting the trigger to the next minute with respect to the current time. This ensured minimal delay when testing. The notification is clearly displayed, regardless of whether the app is running in the background and regardless of what state the device is in. This functionality was manually tested using the simulators that XCode provides.

Feelings Logging:
The main testing mechanism for the logging feature was to type a word or message and save it to the log. Even after closing and reopening the app, the word/message still appears in the logger. Clicking on a message successfully allows the user to update the message or delete it. After manually testing the functionality of the logger, all functionality of this feature is working as expected.

Features We Did Not Implement
Library (Stretch Goal):
The library page was a suggested visual board displaying all the lessons learned in an artful way, where books representing certain lessons would be filled in after the completion of those lessons. This was a stretch goal and was to be implemented if we had extra time after completing the more important parts of the app and finishing the testing.

Calendar (Revised by Client):
In the beginning, the clients wanted us to create a calendar where the user could log their feelings each day and look back to previous days. After talking with them more, the clients decided to pursue a feelings tracker that is not dependent on a calendar for functionality due to time constraints. Instead, they decided on a logger that was similar in feel and appearance to the iOS Notes app.

Login (Canceled by Client):
Originally the team suggested to the client that we could put in a login page in order to view lessons learned on multiple devices. The client later decided that we should not implement this because it might make it difficult for some elderly people to remember their usernames and passwords.

Future Work
A large portion of the future work that can be done on the application involves implementing features that we were unable to complete within the time frame of this semester.

Library page including a visual of lessons completed by the user
A specialized graphic could be created to track the user's lesson progress in a creative and artistic way. The library page would be stylized as a library containing books or other artful graphics. The individual lessons would be able to be clicked in order to revisit that lesson.

User login prior to entering the app
There could be a login page where users could enter their username and password in order to access their lesson learned progress across multiple devices. A database would be used to store the usernames and passwords.

Adding a calendar as a visual component to view one's feelings
A page with an interactive physical calendar could be created. Pressing on an individual day on the calendar would bring up a page that would allow users to input, edit, and view their feelings on any issues on that given day. There could also be a prompt at the beginning of every session asking users to log how they are feeling that day, adding their answers to the calendar.

Make application available on multiple platforms

In addition to being on the iOS app store, this app could also be developed for Android and put on the Play Store. The features would all be the same, and the lessons would be transferable between the two platforms when a user logs in.

Lessons Learned

1. Accessibility within an educational application is extremely important. If the content of the application cannot be reached by varying audiences, the purpose of the application loses its value.

2. Learning XCode in one semester is difficult without background knowledge in app development. We had a difficult time implementing certain features because a new version of iOS and XCode was released this year, and most of the documentation found was irrelevant and invalid for iOS14.

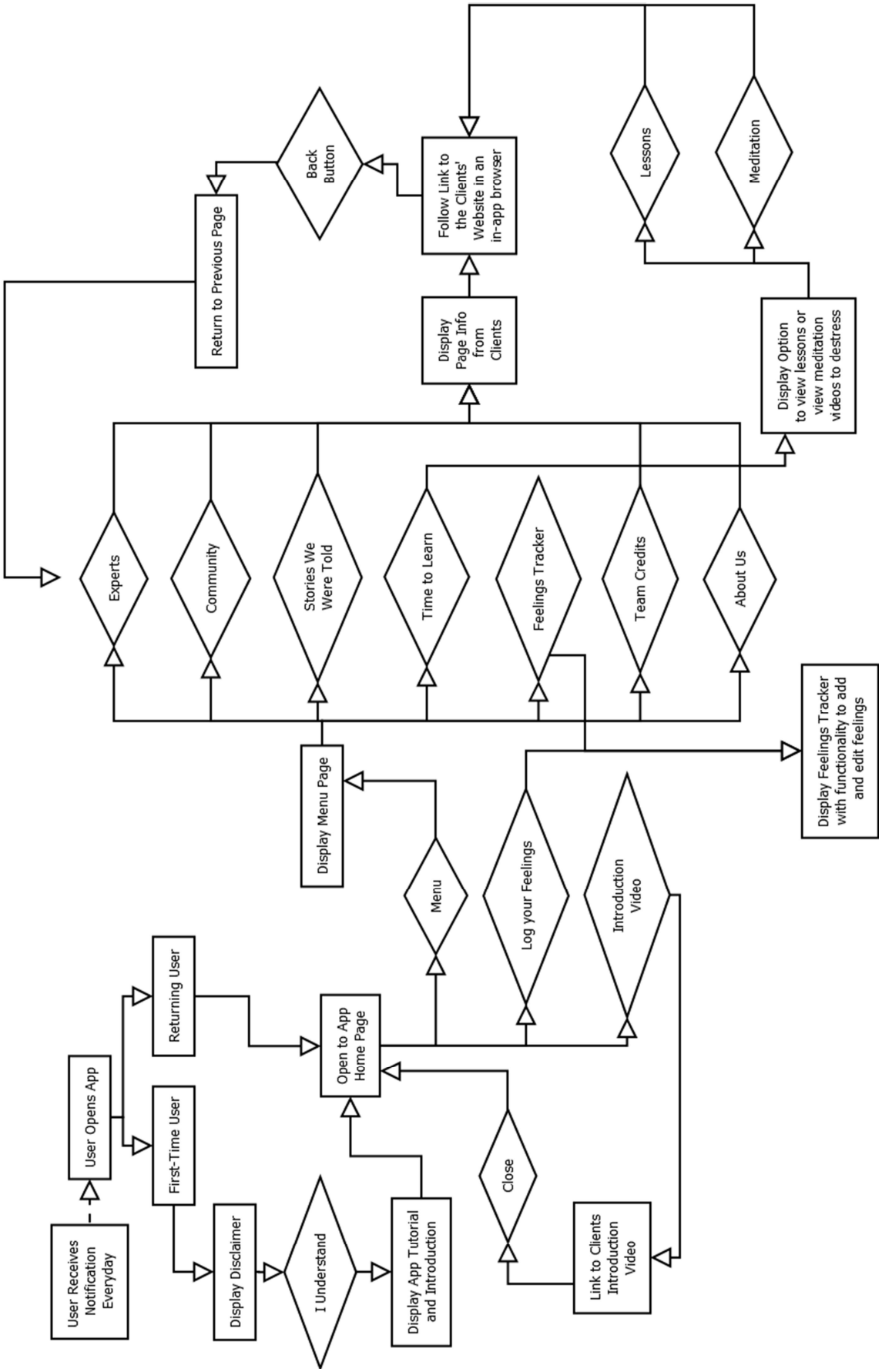
3. What the client says and what the client envisions are two completely different things. Balancing the shifting needs of the clients and our own ability to work on the product is extremely important to its success.

4. Basic application features (navigation, notifications, etc.) are easily implemented in XCode using built in templates, but modifying them to fit the specific needs of the application can be a lot more difficult as there are very few resources on how to do so.

5. The iOS device emulator built into XCode is very useful for testing features such as app navigation, notification alerts, font sizing/readability on different devices, and many other front-end features of the application.

Appendices

Appendix A1: App Flowchart




The Stories We Never Questioned
Menu

Menu:

The Experts:

We are not experts on race issues. There are many amazing people who have devoted their lives to studying race and race issues. Throughout this study we will be referring you to people who are experts.

Click here to go to our website for a masterlist of all the experts we reference.

Community Page

Growing your mind and learning are only the first step of a long journey! I am sure you have heard people say, "no justice, no peace." And getting involved in a community is the best way to promote justice.

click here to go to the community page of our website.

Stories:

We all grew up hearing stories, some of them overt, some of them subtle. All of them deeply woven into the fabric of who we are as a society and as a community. These stories were meant to tell us who we are and who others are, how to stay "safe" and how to contribute.

Not all of these stories were bad or wrong. Many were wise and many had good intentions, but many of them also need to be examined and questioned because some of them are not fair, and some of them have caused a lot of anguish.

The following will take you to the page on our website where you can choose which story to consider.

click here to continue.

Daily Thoughts Tracker

Day 1: Curious

Day 2: Offended

Day 3:

Day 4

Day 5

Day 6

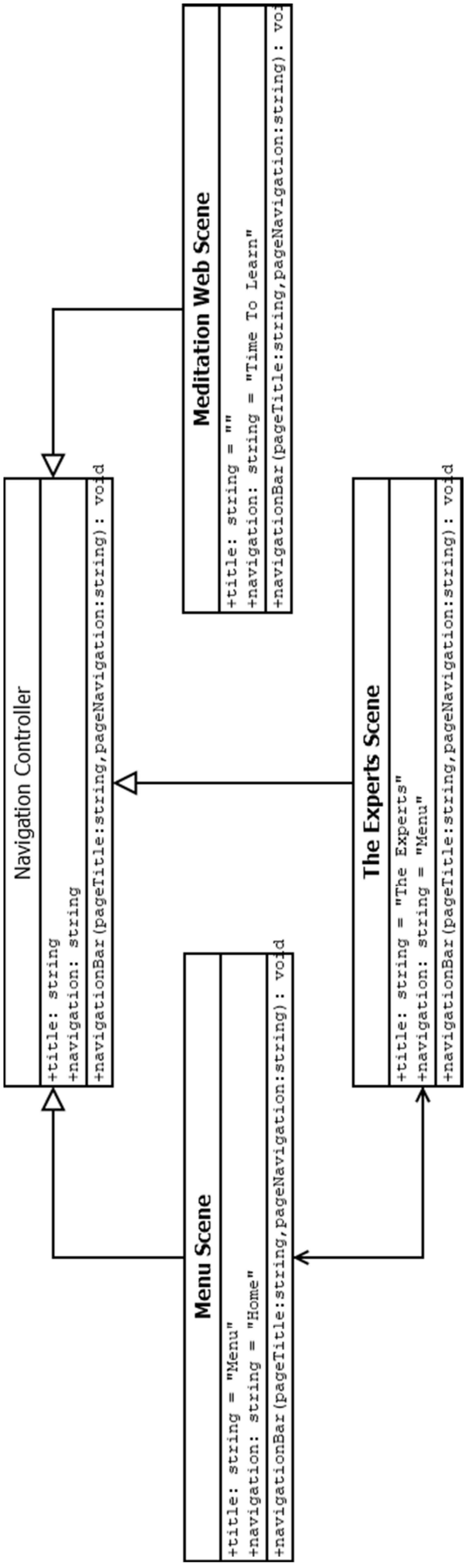
Day 7

.
.
.
.

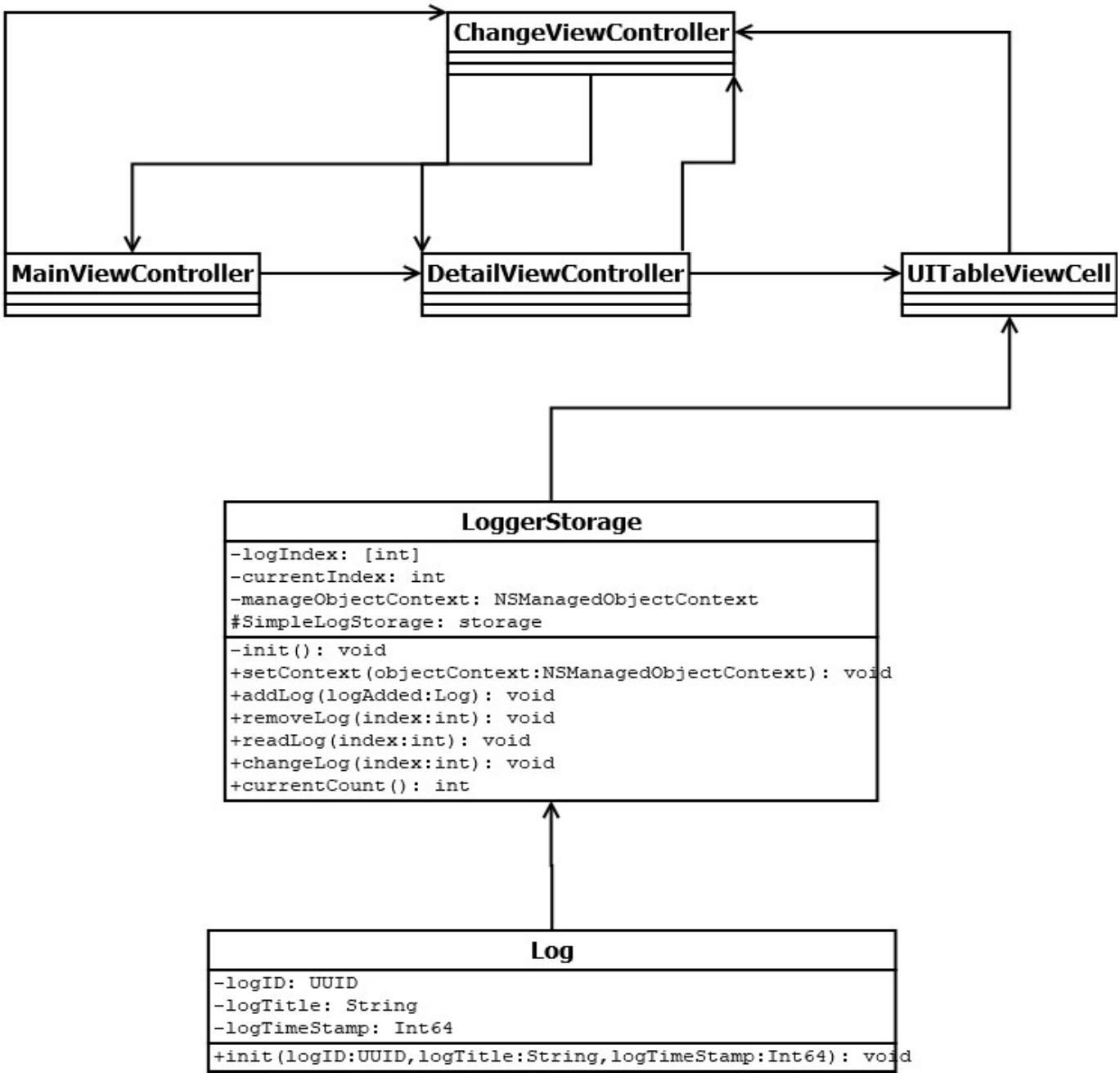
How are you feeling today?

(enter text here)

Appendix A3: Navigation UML



Appendix A4: Feelings Tracker UML



Appendix A5: UILabelVariableDevice Implementation

```
import UIKit

class UILabelVariableDevice: UILabel {

    @IBInspectable
    var fontSize: CGFloat = 0 {
        didSet {
            self.adjustFontSize(size: fontSize)
        }
    }

    func adjustFontSize(size:CGFloat) {
        let fontName = self.font.fontName
        var newFont: UIFont?
        let bounds = UIScreen.main.bounds
        let width = bounds.size.width
        print(width)
        print(fontName)
        switch width {
        case 320.0: //iphone 4s iphone se 1st gen and ipod touch 7th gen
            newFont = UIFont(name: fontName, size: size * 0.81)
            self.font = newFont
        case 375.0: //iphone se 2nd gen, iphone 7/8, iphone 11 pro, 12 mini
            newFont = UIFont(name: fontName, size: size * 0.95)
            self.font = newFont
        case 390.0: //iphone 12, 12 pro
            newFont = UIFont(name: fontName, size: size)
            self.font = newFont
        case 414.0: //iphone 7+/8+, iphone 11, 11 pro max
            newFont = UIFont(name: fontName, size: size * 1.05)
            self.font = newFont
        case 428.0: //iphone 12 pro max
            newFont = UIFont(name: fontName, size: size * 1.10)
            self.font = newFont
        case 768.0: //ipad 9.7"
            newFont = UIFont(name: fontName, size: size * 1.55)
            self.font = newFont
        case 810.0: //ipad 8th gen
            newFont = UIFont(name: fontName, size: size * 1.60)
            self.font = newFont
        case 820.0: //ipad air 4th gen
            newFont = UIFont(name: fontName, size: size * 1.65)
            self.font = newFont
        case 834.0: //ipad 10.5", 11"
            newFont = UIFont(name: fontName, size: size * 1.68)
            self.font = newFont
        case 1024.0: //ipad pro 12.9"
            newFont = UIFont(name: fontName, size: size * 1.70)
            self.font = newFont
        default:
            break
        }
    }
}
```

Appendix A6: UIButtonVariableDevice Implementation

```
import UIKit

class UIButtonVariableDevice: UIButton {

    @IBInspectable
    var fontSize: CGFloat = 0 {
        didSet {
            self.adjustFontSize(size: fontSize)
        }
    }

    func adjustFontSize(size:CGFloat) {
        let fontName = self.titleLabel?.font.fontName
        var newFont: UIFont?
        let bounds = UIScreen.main.bounds
        let width = bounds.size.width
        print(width)
        switch width {
        case 320.0: //iphone 4s iphone se 1st gen and ipod touch 7th gen
            newFont = UIFont(name: fontName!, size: size * 0.9)
            self.titleLabel?.font = newFont
        case 375.0: //iphone se 2nd gen, iphone 7/8, iphone 11 pro, 12 mini
            newFont = UIFont(name: fontName!, size: size * 0.95)
            self.titleLabel?.font = newFont
        case 390.0: //iphone 12, 12 pro
            newFont = UIFont(name: fontName!, size: size)
            self.titleLabel?.font = newFont
        case 414.0: //iphone 7+/8+, iphone 11, 11 pro max
            newFont = UIFont(name: fontName!, size: size * 1.05)
            self.titleLabel?.font = newFont
        case 428.0: //iphone 12 pro max
            newFont = UIFont(name: fontName!, size: size * 1.10)
            self.titleLabel?.font = newFont
        case 768.0: //ipad 9.7"
            newFont = UIFont(name: fontName!, size: size * 1.55)
            self.titleLabel?.font = newFont
        case 810.0: //ipad 8th gen
            newFont = UIFont(name: fontName!, size: size * 1.60)
            self.titleLabel?.font = newFont
        case 820.0: //ipad air 4th gen
            newFont = UIFont(name: fontName!, size: size * 1.65)
            self.titleLabel?.font = newFont
        case 834.0: //ipad 10.5", 11"
            newFont = UIFont(name: fontName!, size: size * 1.68)
            self.titleLabel?.font = newFont
        case 1024.0: //ipad pro 12.9"
            newFont = UIFont(name: fontName!, size: size * 1.70)
            self.titleLabel?.font = newFont
        default:
            break
        }
    }
}
```

Appendix A7: Feelings Tracker Implementation

```
import CoreData

class ReallySimpleNoteStorage {
    static let storage : ReallySimpleNoteStorage = ReallySimpleNoteStorage()

    private var noteIndexToIdDict : [Int:UUID] = [:]
    private var currentIndex : Int = 0

    private(set) var managedObjectContext : NSManagedObjectContext
    private var managedContextHasBeenSet : Bool = false

    private init() {
        // we need to init our ManagedObjectContext
        // This will be overwritten when setManagedContext is called from the
view controller.
        managedObjectContext = NSManagedObjectContext(
            concurrencyType:
NSManagedObjectContextConcurrencyType.mainQueueConcurrencyType)
    }

    func setManagedContext(managedObjectContext: NSManagedObjectContext) {
        self.managedObjectContext = managedObjectContext
        self.managedContextHasBeenSet = true
        let notes =
ReallySimpleNoteCoreDataHelper.readNotesFromCoreData(fromManagedObjectContext
: self.managedObjectContext)
        currentIndex = ReallySimpleNoteCoreDataHelper.count
        for (index, note) in notes.enumerated() {
            noteIndexToIdDict[index] = note.noteId
        }
    }

    func addNote(noteToBeAdded: ReallySimpleNote) {
        if managedContextHasBeenSet {
            // add note UUID to the dictionary
            noteIndexToIdDict[currentIndex] = noteToBeAdded.noteId
            ReallySimpleNoteCoreDataHelper.createNoteInCoreData(
                noteToBeCreated:          noteToBeAdded,
                intoManagedObjectContext: self.managedObjectContext)
            // increase index
            currentIndex += 1
        }
    }

    func removeNote(at: Int) {
        if managedContextHasBeenSet {
            // check input index
            if at < 0 || at > currentIndex-1 {
                return
            }
            // get note UUID from the dictionary
            let noteUUID = noteIndexToIdDict[at]
            ReallySimpleNoteCoreDataHelper.deleteNoteFromCoreData(
                noteIdToBeDeleted:        noteUUID!,
                fromManagedObjectContext: self.managedObjectContext)
            // update noteIndexToIdDict dictionary
            // the element we removed was not the last one: update GUID's
            if (at < currentIndex - 1) {
                // currentIndex - 1 is the index of the last element
```

```
                // but we will remove the last element, so the loop goes only
                // until the index of the element before the last element
                // which is currentIndex - 2
                for i in at ... currentIndex - 2 {
                    noteIndexToIdDict[i] = noteIndexToIdDict[i+1]
                }
            }
            // remove the last element
            noteIndexToIdDict.removeValue(forKey: currentIndex)
            // decrease current index
            currentIndex -= 1
        }
    }

    func readNote(at: Int) -> ReallySimpleNote? {
        if managedContextHasBeenSet {
            // check input index
            if at < 0 || at > currentIndex-1 {
                return nil
            }
            // get note UUID from the dictionary
            let noteUUID = noteIndexToIdDict[at]
            let noteReadFromCoreData: ReallySimpleNote?
            noteReadFromCoreData =
ReallySimpleNoteCoreDataHelper.readNoteFromCoreData(
                noteIdToBeRead:           noteUUID!,
                fromManagedObjectContext: self.managedObjectContext)
            return noteReadFromCoreData
        }
        return nil
    }

    func changeNote(noteToBeChanged: ReallySimpleNote) {
        if managedContextHasBeenSet {
            // check if UUID is in the dictionary
            var noteToBeChangedIndex : Int?
            noteIndexToIdDict.forEach { (index: Int, noteId: UUID) in
                if noteId == noteToBeChanged.noteId {
                    noteToBeChangedIndex = index
                    return
                }
            }
            if noteToBeChangedIndex != nil {
                ReallySimpleNoteCoreDataHelper.changeNoteInCoreData(
                noteToBeChanged: noteToBeChanged,
                inManagedObjectContext: self.managedObjectContext)
            }
        }
    }


    func count() -> Int {
        return ReallySimpleNoteCoreDataHelper.count
    }
}
```

Appendix A8: Coding Conventions

Opening braces appear on the same line as the previous line.

Closing braces appear on their own line.

Title case without spaces for names of files and class names.

Header comment at the beginning of every page of code.