OneTouch Design Doc

CSCI 491 Senior Project

Noah Stull
Will Snyder
Andrew Pasimio Wylie Mickelson

June 6 2025

Contents

1 Product Overview

1.1 Background

The OneTouch app is designed to optimize the note-taking process for its users. OneTouch is tailored to professionals and employees with narrow time frames and limited attention. OneTouch provides customizable note formats to simplify data recording for workers on the job. The application requires only a single button click per note to make recording easy and fast.

1.2 Major Features

1. Customizable note outline creation and editing.
2. Note-taking based on a predefined and saved outline.
3. One-handed accessibility for ease of use.
4. Note and outline organization.
5. Export to pdf service for completed notes.

2 Project Architecture and Design

2.1 Architecture

The architecture can be split into three layers: a UI layer, a Logic layer, and a Data layer. The UI layer will be responsible for the user view, inputs, and interactions. The Logic layer will be responsible for logic, use cases, and interfacing with the Data layer. This layer also handles rules like "don't save empty notes". The data layer will handle fetching from local storage and caching any frequently used notes/sessions. The following diagram (Figure 1) represents a high-level outline of this architecture.

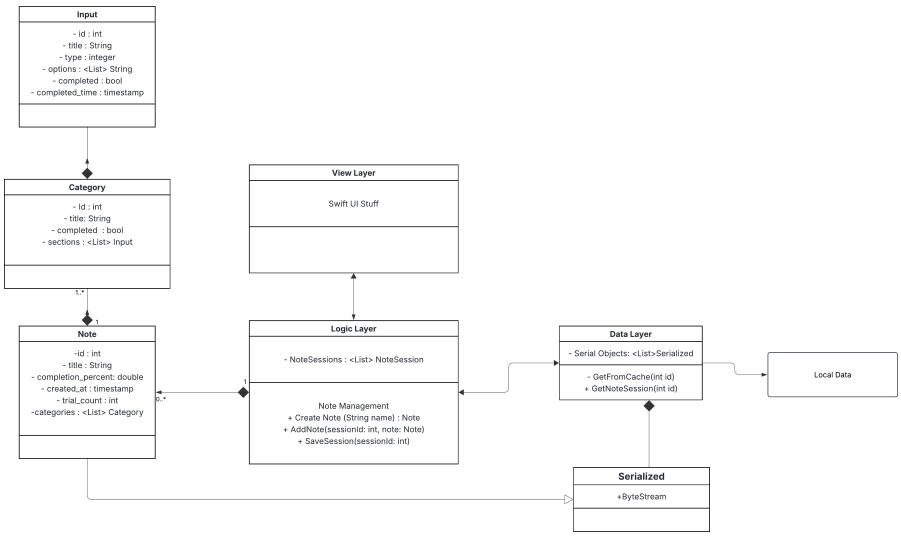

Figure 1: Design Diagram

2.2 Dependencies

1. **Swift** - Main language used to develop iOS applications
2. **Swift Testing** - Easy and automatic testing framework for Swift applications
3. **SWiftUI** - App Frontend
4. **CoreData** - Local data and storage handling

2.3 UI Design

Home Screen The home screen, recent sessions will be stored and ready to go for the user to return to and modify (Figure 2).

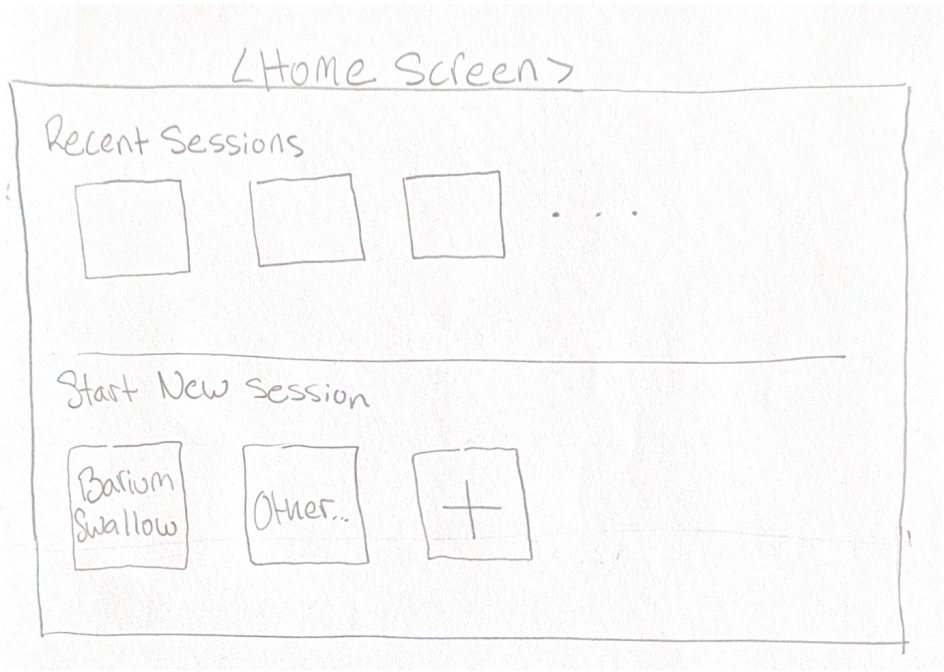

Figure 2: Home Screen

First Trial Screen An example of a note-taking session using the "Barium Swallow" template (Figure 3). The first trial has three categories: consistency, modification, and results. Each section contains a different kind of note. Consistency is a single-question, multiple-choice note with eleven options to be chosen from. After a user is done with the note they will press the arrow to move to the next note section.

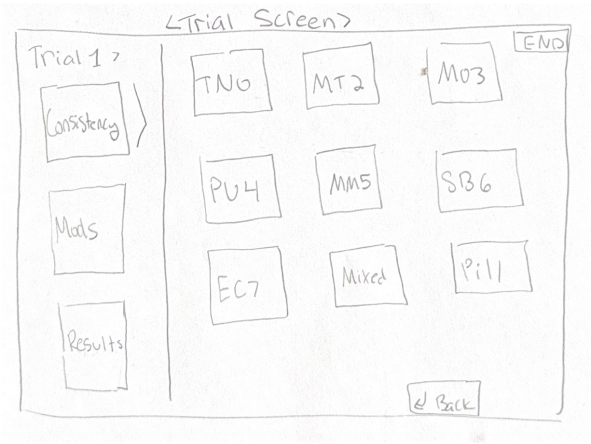

Figure 3: First Trial Screen

Trial Screen 2 The first thing to notice in the next screen (Figure 4) should be that the consistency section has changed to reflect the prior input. In the modification section, the user is offered multiple choices that they can choose one, none, or many of. Since this is our second section, they will also be able to go back to the previous section and modify it, in addition to still progressing forward.

Figure 4: Second Trial Screen

Trial Screen 3 The final note section (Figure 5) for this trial has multiple number scale inputs, which are divided into sections again. Since this is the last section for the trial, the user will have the option to either go back or move on to the next trial. If the user chooses to go back, they can modify the results of the first trial, and moving to "Next Trial" will take the user back to consistency and change from Trial 1 to Trial 2.

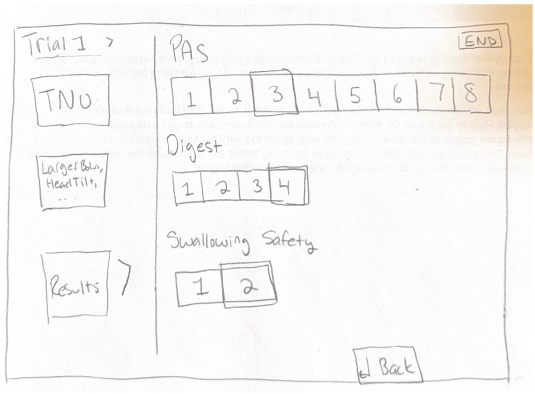

Figure 5: Third Trial Screen

Results Screen If at any moment the user wants to conclude early, they can press the "finish" button in the top right and conclude early. This, along with normal progression, will take you to the results screen. (Figure 6)

The final screen displays all of the details that the user input in a bulleted list, along with their corresponding time stamps. The buttons on the bottom allow the user to go back and modify data again, save and return to the home screen (Figure 2), or export via email. Upon saving, the recent sessions will update to keep the new note in the first slot until a new one is written. Choosing to mail the document will convert the results page into an equivalent pdf to be sent to the recipient. The format of this sheet will depend on how the user customized the trials and session.

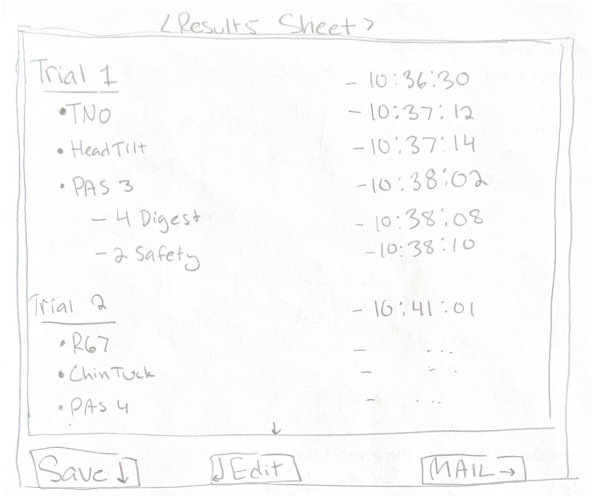

Figure 6: Results Page

2.4 Testing and Verification

Swift Testing will be used for unit testing the logic layer of the app. Some unit tests that will be implemented are as follows:

- Option selection: Ensure that selecting an option works correctly, along with selecting and unselecting an arbitrary number of times.
- Multiple choice selection: If a note has multiple options selected (and it's valid) are both still selected.
- Input Completion Test: Create an input, set completed to true, set completedTime to the current time, and assert that completed is true and the time is not null.
- Note Creation: Create a note and assert that all values in the constructor are assigned properly.

2.5 Dependencies

- Swift for the main part of frontend and backend logic implementation
- Swift Testing
- SwiftUI for visuals and messaging/data exportation
- CoreData - IOS data management framework for local storage

2.6 Frontend Technical Details

As shown in the dependencies section, the frontend of our app will almost entirely be written in Swift to suit our client's needs when it comes to developing an iOS app. We will also take advantage of SwiftUI, which is a Swift framework that will be used specifically to build the user interface part of our application.

3 Data Layer

3.1 Data Management

Due to HIPAA compliance, no data will be sent or stored on any remote servers. Instead, all notes and models will be saved directly to the hosting device. Therefore, the "backend" will not be hosted on a server itself. It will be part of the same package containing the other layers. To accomplish this, we will have a backend data management class containing all relevant data retrieval and storage operations, which will act as a kind of "mock" backend.

Swift has a very helpful framework, CoreData, which is used to manage data in iOS applications. It allows developers to store, retrieve, and manipulate data in an object-oriented manner. It allows for persistent local storage, which is perfect for this use case. CoreData is what we will use for all operations relating to user data.

3.2 Saved Object Structures

Figure 7: Saved Object Structures

The structure of the saved objects can be seen in Figure 6, with a detailed description of each as follows:

- Model
  - Models are the predefined structure that notes will follow when a user begins a session.
  - The outline variable will contain a Note object without any user input, essentially just a blank slate.
  - Last-viewed and created-at timestamps allow for easy sorting of model outlines on the UI layer.
- Note

  - Notes are the implementation of a Model object.
  - Trial-count determines the amount of trials to complete on an individual note. One trial is a complete pass through all of the categories and inputs.
  - Notes can contain one or more categories

- Category
  - Categories are one collection of Input objects. These objects are just a container to store relevant information together.

- Input
  - Inputs are the smallest piece of information on a Note. These contain information about what the user actually entered during a session.
  - Options are the predetermined values a user can choose from to save during a session. For every input, there will always be an "other" option where the user can add something that was not predefined.
  - Completed-time allows for accurate and specific information regarding when a trial was completed. This helps with examining trials once the entire note is complete.

4 Data Exportation

Completed notes will have an option to export to pdf, where the saved note data will be converted into a simple, readable text format as a pdf file. From here, data will be exported directly to a saved user email via the Swift MFMailComposeViewController. This will allow the app to set the contents of the email to contain the notes and address it to the user, the function will look similar to this:

```
func sendEmail() {
    if MFMailComposeViewController.canSendMail() {
        let mail = MFMailComposeViewController()
        mail.mailComposeDelegate = self
        mail.setToRecipients(["user@theiremail.com"])
        mail.setMessageBody("<p>NOTE CONTENTS HERE</p>", isHTML: true)
        present(mail, animated: true)
    } else {
        // show failure alert
    }
}

func mailComposeController(_ controller: MFMailComposeViewController, didFinishWith result:
MFMailComposeResult, error: Error?) {
    controller.dismiss(animated: true)
}
```