


Spring 2023

Reed Log: Application for Oboists

Michał Cieślik
Bard College

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2023

 Part of the [Graphics and Human Computer Interfaces Commons](#), [Music Education Commons](#), [Music Performance Commons](#), and the [Software Engineering Commons](#)



This work is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License](#).

Recommended Citation

Cieślik, Michał, "Reed Log: Application for Oboists" (2023). *Senior Projects Spring 2023*. 198.
https://digitalcommons.bard.edu/senproj_s2023/198

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Spring 2023 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.

Reed Log: Application for Oboists

Senior Project Submitted to
The Division of Science, Math, and Computing
of Bard College

by
Michał Cieślik

Annandale-on-Hudson, New York
May 2023

Acknowledgments

I would like to thank my teachers for providing me with the knowledge that made me into the musician that I am today and inspired the work of this paper. The support that you've provided me throughout my education has been invaluable and inspiring.

I would also like to thank Rose Sloan for advising me throughout this enormous project. I don't think I could complete this without your support, and I am eternally grateful for that.

Finally, I would like to thank all of the oboists that helped me test this application, mainly Elizabeth Perez-Hickman and Alex Norrenberns, for actively using the application throughout its development, and suggesting changes and improvements to the application.

Table of Contents

Abstract.....	6
Introduction.....	8
Introduction to Oboe Reeds.....	8
Cane Processing.....	9
Cane Tying.....	10
Reed-Making Goals.....	11
The Oboe Bible.....	12
Reed Tests.....	13
Reed Qualities.....	14
Purpose and Functionality.....	15
Functionality.....	16
Uses for the App.....	16
JSON.....	17
App Implementation.....	18
SwiftUI vs UIKit.....	18
General SwiftUI View Structure.....	20
Reed Log's Structure.....	20
MVVM.....	21
Views.....	22
Views as Code.....	24
Views in Practice.....	26
Property Wrappers.....	27
Using Views and Method Calls.....	28
ViewModels within Views.....	29
View Models.....	31
VMs and Views.....	31
Property Wrappers Revisited.....	32
VMs and the Model.....	33
Context Introduction.....	33
NSFetchedControllerDelegate.....	34
controllerDidChangeContent().....	36
Storing New Data.....	37
Model.....	38
Core Data Revisited.....	38
Entities and Data Types.....	39
CoreDataClass and CoreDataProperties.....	41
DataController Class and NSPersistentContainer.....	42

App Evaluation.....	43
Speed Tests.....	44
Results.....	45
Memory Tests.....	46
Results.....	47
Conclusion.....	49
Future Projects.....	49
Studies Involving the Reed Log.....	49
Commercial Uses.....	50
Conclusion.....	51
Bibliography.....	53

Abstract

Reed-making is a crucial part of the art of oboe playing. The quality of the reed predetermines most (if not all) of how the instrument sounds and the art of making reeds is a significant part of one's musical education. As Julia Gjebic puts it: "Professional oboists spend more time making reeds than they do playing the oboe" Each step in the reed-making process reveals a lot of information about the final qualities of the reed, with even the most minor decisions in the early process having significant ramifications later on. Sadly, because of the number of variables present, a lot of said information is lost in the process, making the process more complicated than it already is.

The Reed Log was designed to address this problem by creating a pocket data analysis tool for oboists in the form of an app, allowing oboists can digitally monitor and analyze the data about their reeds. The app allows the user to store reed data in a standardized format and easily access said data. It also provides the user with a built-in analysis screen that provides the user with pertinent information about one's reeds, as well as built-in JSON support for use within studies. The application was tested for speed and memory usage and is currently available on the App Store.

Introduction

Introduction to Oboe Reeds

A number of woodwind instruments use reeds to produce a sound. The family of reed instruments has two distinct groups: double-reed and single-reed instruments. The oboe, just like the bassoon, is part of the double-reed instrument group, while the single-reed instrument group includes instruments like clarinet or saxophone. The basic structure of a double reed consists of two pieces of a specific variety of grass called *Arundo donax* tied together and scraped with a knife to produce a specific pitch.

While single-reed instruments can purchase factory-made reeds even at the professional level, double-reed instruments do not enjoy the same benefit — double reeds need to be highly personalized to the player and drastically change with use or age, atmospheric pressure, or humidity changes. Because of the volatility of reeds, oboists (and other double-reed players) have to make their reeds from scratch.

Because of the complexity of the beforementioned process, I made the Reed Log application as an aid in the reed-making process. The app stores information about one's reeds, which helps systematize one's reed-making. It also provides the user with statistical analysis on their reeds and provides built-in JSON support for use in reed-making studies.

To better understand the concepts central to the app's existence, this paper will attempt to summarize the process of creating a reed from scratch, which each oboist follows when making a single reed. It will also show which information gets recorded about one's reed within the app and for what purpose. While the specifics might differ between players, the general process still applies, if only in spirit.

Cane Processing

An oboist's work on a reed starts with a small tube cut from the stem of *Arundo Donax*, also called a giant reed. This piece of cane is then dried and aged to prepare it for use, changing its color from green to yellow. Said color and several other characteristics of the cane are used when selecting which pieces to use, and it is a critical step in making reeds. Two data points are extracted from this process: the cane's **type** and **diameter**. The **diameter** is measured with a radius gauge when the cane is still in tube form, and it has the most significant influence on the opening size of a given reed. On the other hand, the cane **type** is simply the brand from which the cane was purchased. Cane can vary significantly based on where or when it was grown, so cane will often share characteristics with the cane of the same brand and batch. Because of this fact, the app abstracts all of the other characteristics of the cane into the **type** category to streamline the usage of the application.

After selecting the piece of cane with the desired qualities, an oboist needs to split the tube so that the resulting one-third of the tube is as straight as possible. Usually, only one piece of cane can be retrieved from a single tube, although more pieces can occasionally be found. Next, an oboist must soak the material in warm water for an extended time. This treatment makes the material more pliable, ensuring it does not crack. After soaking, the cane is cut to a specific length and planed flat with a regular hand planer, after which it is ready to be gouged. The gouging process involves thinning a piece of cane with a gouging machine. This process is exact, usually taking off around 0.08-0.10mm of cane at a time, with significant consequences on the reed's qualities later in the reed's life. Many different gouging machines with different blade architectures profoundly impact the qualities of the reed, and oboists often choose to use specific gouging machines on reeds for musical reasons. For example, gouges with thicker measurements

will force one to scrape the cane deeper than a thinner gouge, making the reed more cushioned but quiet. The app records the gouging machine used on a specific reed under the cane's **gouge**.

After gouging a piece of cane, the piece of cane is then shaped from rectangular to reed-shaped using a shaper tip (or a shaping machine). This process is done by first folding the cane in half and placed on a shaper tip. An oboist then removes the excess cane from the sides of the shaper, making the piece of cane curved and ready for tying. Shaper tips influence what shape the cane will take after tying it onto a staple, and it is recorded under the cane's **shape** category.

To check the internal architecture of the reed defined by the gouge and shape, an oboist will usually measure the piece of cane immediately after shaping the reed. Variations in these measurements on the smallest of scales can significantly impact the reed's success, with the relationships between the measurements needing to be accurate within 0.01 - 0.02mm. These measurements give an incredible insight into the reed's architecture and can reveal issues arising from previous steps. Besides revealing problems in the reed, the measurements can help an oboist modify their scrape depending on said measurements. The app records these parameters under the cane **measurement** section.

Cane Tying

After shaping a piece of cane, the cane is ready to be tied onto a **staple**, or a small brass or silver tube partially wrapped in cork, using nylon thread. There are many different types of **staples** with differing thicknesses, architectures, and materials of the tube, which all affect the reed's performance. The critical relationship in this stage of reed-making is finding a good **tie length** for a specific staple and shape used, or the length measured from the bottom of the staple

to the top of the piece of cane. The **tie length** determines what part of the shape will be under the string and what part will form the outside of the reed, considerably influencing how these parameters interact. The application records both **staple** and **tie length** under the tying information section and other convenience information like **staple ID** or **thread color**. When a piece of cane is tied onto a staple, it is classified as a blank reed, at which point it is ready for scraping.

Reed-making Goals

So far, the reed-making process has consisted of several quantifiable and standard steps. Each step relied on experimentation and an analytical approach, easily translating into identifiable and testable quality metrics. Said steps were still based on several subjective decisions, but one could still identify specific metrics that are always positive or negative. The cane and tools one uses have direct and testable effects on a reed's qualities. For example, reeds with extremely uneven measurements in the gouge will most likely be worse than even ones, and an incredibly soft, white, and crooked piece of cane will most likely be worse than a straight, smooth, golden one.

Sadly, the scraping process does not lend itself to such an approach. While there are specific spots one can scrape which will have consistent and testable results, the scrape of the reed can vary wildly between reeds and still play just as beautifully. **In other words**, a lot more is left to personal tastes, so a sense of objectivity is not always applicable. The success of the reed relies on how good a reed sounds. While that is the truth behind reed quality on all levels of reed-making proficiency, the definition of reed quality does not specify what a "good-sounding reed" actually is.

Oboists worldwide have been trying to perfect the art of reed making by striving closer and closer to their ideal sounds. Because of said artistic freedom in one's definition of a good sound, several reed-making processes were developed, with differing sound styles developing in tandem. While reeds drastically differ from person to person, even with similar educations, these styles can be distilled into two general groups: short-scrape and long-scrape reeds. Short scrape reeds are most commonly found in European countries, while long scrape reeds are found in America. Because of my educational background and sound preferences, the app will focus on the long scrape approach to reed-making.

The Oboe Bible

Graham Salter's book *Understanding the Oboe Reed* (Salter, 2018) is a foundational resource for oboists of all varieties. It is a 552-page exploration of oboe reed styles, detailing techniques and findings from many past and present pioneers in the field. It is treated by many as the sacred text of reed making, and for that reason, the book has been nicknamed "the oboe bible."

Although a bit cheeky, the nickname demonstrates this book's immense influence on oboists since it was published. The section most influential within the American oboe sphere is *An Objective Approach to Reed-Making by Elaine Douvas and Linda Strommen*, which is the foundation and necessary prerequisite for this application. It concerns itself with classifying objective tests that determine whether a reed is successful or not. It is highly systematic and relies on an oboist making a sound on a reed outside the oboe, which limits one's impulse to compensate for the reed's flaws. Furthermore, the tests provide one with a metric that is not reliant on one's artistic sensibilities, instead relying on the presence of specific qualities that are present regardless of who is playing them.

These reed tests are still not entirely objective, for one relying on one's mouth and lip shape, but they provide a testable and replicable foundation to build from. The tests mainly rely on reed performance, favoring creating a reed that one feels comfortable in rather than a reed that one fights. The focus is shifted from how the reed sounds to what the reed can do, like start on time, or have a range of dynamics, letting one decide on their definition of "good sound."

Reed Tests.

The method in said book classified four tests, with versions on the reed alone and while playing the oboe normally. These tests are not as simple as using tools, with immense complexity built into them and requiring multiple years of study to master them fully. Therefore, this essay will only briefly discuss each test and qualify what qualities of the reed it is testing for.

The first test is the "aspirated attack" test, where an oboist blows lightly into a reed and then increases one's airspeed until a sound appears. This test mainly tests the **response**, or how easy it is to start a sound on a reed. The second test is the "peep" test, where an oboist repeatedly plays a sound on a reed in a normal playing position, starting and stopping the sound with one's tongue. This test checks the presence of multiple qualities in a reed, the biggest ones being **pitch floor** or what pitch a reed plays at when playing in a comfortable position. This test also gives insight into the **tone** of the reed, revealing whether the sound produced is clear and ringing. The third test is called the "glissando test." It is done by playing a sound at a comfortable position and slowly pushing the reed out with one's lips, which tests the **flexibility** of a reed. Lastly, the "crow" test is done by putting the entirety of the reed that vibrates into one's mouth and making a sound, essentially seeing how the reed vibrates with no lip support. This test gives further insight into the **tone** of the reed, revealing how the parts of the reed vibrate as a whole. (Salter, 2018)

These tests implicitly define the exact qualities that a reed possesses in order to be considered successful. They provide a set of objective metrics that can be used while making reeds to streamline the process. It provides a benchmark of quality that provides consistency of results and a semi-objective metric of the reed's success based on its qualities.

Reed Qualities.

While these tests did not invent the qualities discussed in earlier sections, they made them more tangible as concepts, helping to classify them through objective tests rather than one's preferences. In other words, rather than relying on qualitative statements and feelings about how a reed sounds when played to judge its quality, these qualitative sentiments become quantifiable through tests, separating them into concepts that can be discussed strictly and universally.

Within the app, each reed has a **questionnaire** section attached at the bottom of the parameters of the reed. Several qualities defined by the tests were omitted, both for brevity and ease of use of the app and because they are too reliant on the quality of one's scraping skills rather than any of the internal parameters of the reed. When first learning how to make reeds, the scraping aspect of reeds affects reed success the most. When one does not have the consistency of scrape, the internal workings of a reed are not as much of a concern since students struggle to scrape a reed with sufficient detail. As both one's scraping technique and one's standards for reeds improve, the scraping of a reed becomes a lot more consistent and standardized, and the internal architecture of the reed becomes critical. The **response** and **resistance** of a reed stop being as much of an issue as other metrics, including tone **depth** and **ring** or **stability** and **flexibility**. **Pitch** is always a concern, with both the scrape and internal attributes of the reed influencing it greatly.

Purpose and Functionality

As seen by the length of the previous sections, the qualities of a specific reed rely on several prerequisite reed parameters. Each step in the process determines a significant portion of what the reed will sound like before the reed is anywhere near an oboe. Countless pages could be spent exploring the intricate relationships between these parameters and countless more on how they relate to the scraping process. However, one first needs to keep track of these parameters to start that conversation.

When working on reeds without writing anything down, a staggering amount of data that critically affects the qualities present in the reed is often discarded, making the process much more difficult. When one's reed-making output becomes larger, writing all of this information down in a notebook about each reed made becomes tedious and impractical. Besides taking too much time, referencing information and drawing educated conclusions is nearly impossible.

Technology has been a part of our lives for a long time now, bringing benefits in performance and accuracy in many parts of our lives. While apps made for classical musicians exist, with many tuning apps or practice logs being used regularly, the role of technology in one's individual musical life is still limited. The study *Technology Use and Attitudes in Music Learning* explored the use of technology within personal and educational settings. It collected data through a survey to explore how apps like tuners, metronomes, or audio recorders were used in a one-on-one educational setting. It found that there are immense opportunities “for technology to take a greater role in improving music learning through enhanced student-teacher interaction and by facilitating self-regulated learning.” (Waddell & Williamon, 2019)

The Reed Log app was made as an additional tool for oboists to use in their careers. It attempts to streamline the process and save as much information about one's reeds as possible while reducing the tedium of the traditional method. Students often dread writing said information in a notebook and said the app would provide a faster and better alternative. It also provides teachers with a perfect log of their students' reeds, making spotting issues with the students' reeds a lot simpler.

On top of streamlining the process of writing down information about reeds, the interactive medium of an app gives the user the ability to do quick searches and analyses using the app. The data collected can be used in many powerful ways to help both the student and the professional reed maker. The app should be considered a pocket data collection and analysis tool for oboists. Said functionality can be helpful both in lessons and in one's personal education or professional use.

Functionality

The Reed Log's functionality hinges on the central purpose of the app: mainly storing information about oboe reeds. Because of that, most of the app's functionality is centered around adding, deleting, and editing reed objects within the app, although more functions are present. To be precise, the app facilitates storing reeds within distinct reed boxes and displays reeds based on a number of different parameters, like reed stage within the main screen, staple ID within the search functionality, or existence in a reed box through the reed boxes themselves. On top of said functionality, the application also computes statistical analysis on the collected data and provides functionality to convert said data into JSON.

Uses for the App

While keeping a log of one's reeds are not a new concept in oboe reed making, the ease of access to the information immediately improves one's analysis of one's reeds. The Reed Log app was specifically designed to help fix many issues that can arise when data about reeds is discarded. These issues range from fundamental issues of equipment to conceptual failings with the scraping of the reed itself and can benefit both a student and a professional reed maker. The app can immensely help one spot issues with one's gouging machine by spotting inconsistencies within the measurements, and it can help decide which measurements one prefers. In addition, it can help spot configurations between parameters that maximize desirable metrics, allowing one to start predicting what qualities a reed might have very early in the process. The app also can help one spot trends with one's reeds, like one's reeds tending to be sharp and help find causes of said trends.

The benefits of using this app mainly stem from one's ability to reference previous data easily. For example, when one is curious about a specific reed, one simply has to enter the ID of a staple to access every reed that one has ever logged with the said staple, allowing one to compare results directly. On top of ease of access, the app provides analysis functionality that gives several statistics based on the reeds present. Currently, the app provides one's reed box makeup, giving one the idea of how many reeds at each stage one has present and average measurements, loudness, and pitch of the reeds. These metrics can help users spot glaring issues with their reed-making by aggregating data that could not have been easily analyzed by hand.

JSON

As mentioned, the app aims to become a pocket data collection and analysis tool. While the application's analysis screen can reveal important information about one's reeds, the Reed Log also leaves space for deeper analysis. By implementing an option to convert one's reeds into

JSON and export them outside of the app, the app can be used to collect reed data for many studies. The relationships between reed parameters are very complex, and several studies could be conducted to unravel these concepts.

App Implementation

Because of the focus on ease of use and access, much of this project's creation centered around the application's visual aspect. The user needed to use the app with ease to encourage as much data to be collected as possible without introducing tedium back into the process. For that reason, the choice of UI framework was critical in this app's development – one that focused on clarity and ease of use. A UI framework is a set of tools that allow a developer to craft the user interface (UI) in an abstract manner, leaving the actual inner workings of the framework separate from one's app implementation.

The app itself was made using the language Swift and the Xcode UI creator extension called SwiftUI. SwiftUI is one of the two UI frameworks used within XCode, the other being UIKit. It is newer than UIKit and creates the UI programmatically; instead of crafting the visual aspects of the application through visual editors, SwiftUI allows the developer to define the UI through written code. This difference allows one to focus on the relationships between UI items rather than placing them at specific points on the screen, making the application much more transferable between different device sizes.

When starting the development of an application in XCode, one has to decide between these two frameworks as the project's foundation. The two frameworks often work in tandem with each other, though, and one can insert a view made in one into the other freely.

SwiftUI vs. UIKit

The main difference between SwiftUI and UIKit is that the first uses declarative programming, while the second uses imperative programming. Their approaches to creating the UI are completely different, and both are made with different goals in mind. SwiftUI excels with the speed of development and performance, while UIKit gives complete control over what is being shown on screen, giving the developer much more freedom. UIKit accomplishes this by operating with less abstraction and making the developer focus on describing how the program actually operates. In contrast, SwiftUI is centered around creating and using abstracted chunks of code that act like black boxes, which create the UI through the interactions between each other. In other words, “We aren’t making SwiftUI components show and hide by hand, we’re just telling it all the rules we want it to follow and leaving SwiftUI to make sure those rules are enforced.” (Hudson, 2021e) A declarative approach of SwiftUI makes managing and structuring a project significantly easier, allowing for much less time to complete an application.

UIKit is based on the interaction between your own code and a number of Storyboard files. Said files are models of the components present on-screen created through the Interface Builder (IB), a visual editor within XCode. The work within the IB is very similar to those in engines like Unity or Unreal Engine. It is completely separated from any of the Swift code and needs to be connected manually through XCode’s interface to gain the desired functionality.

Because of its imperative approach, UIKit relies on the IB to create most of the UI, relying heavily on a program external to the code written. UIKit is fundamentally “designed around the way Objective-C works” (Hudson, 2021d) and relies on explicit connections between the IB’s visual interface through XCode functionality. The reliance on Storyboards makes the workflow a lot slower and a lot more difficult, and is prone to a number of errors that are based on missing or obsolete connections between the IB and the code.

SwiftUI on the other hand solely relies on code written in Swift, with almost no external inputs from tools like IB, making it a lot easier to structure and complete projects. Because of the ease of use of SwiftUI compared to UIKit, and the growing number of functionality present within SwiftUI itself, it proved to be the better option for this project.

General SwiftUI View Structure

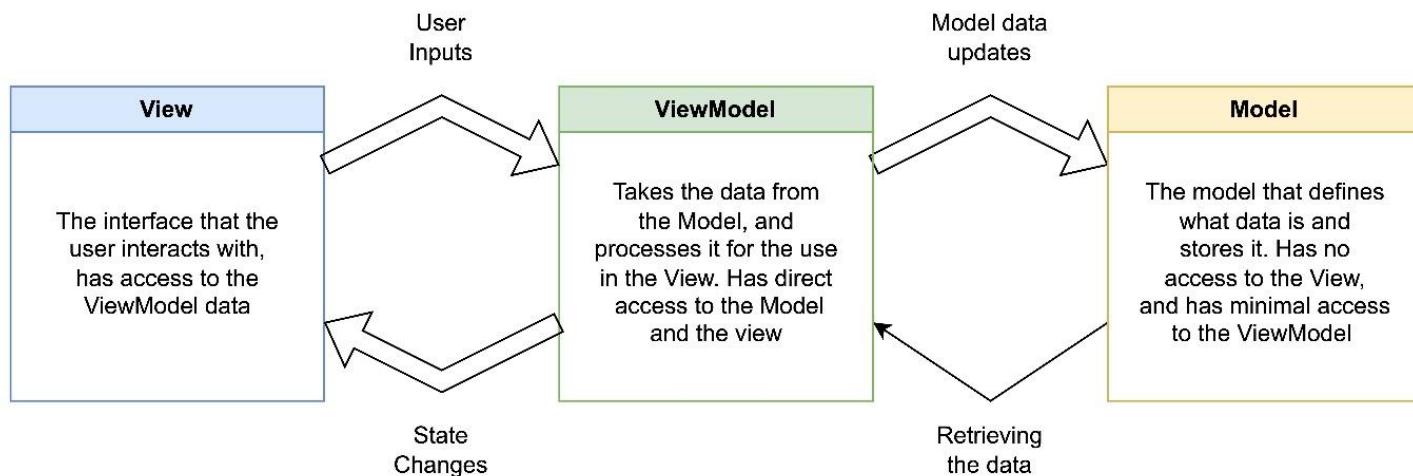
As mentioned before, the creation of all UI elements is completely done programmatically. All of the UI elements seen on screen are black boxes that have different functionalities, either predefined within SwiftUI's library or defined explicitly by the developer. The structure of the app is based on a nested set of structs that display information, facilitate user inputs, and/or host other structs that do the same. Most of these structs conform to the View protocol, and so from now on they will be referred to as Views themselves. Some of these views include views that groups data into containers like HStack, VStack, or List, views that have specific functionalities like Button, Picker, or Slider, or views that one defines when creating the application themselves, like MainView, AddReed, or ReedDetail. In other words, each View is a collection of nested Views which are made to complete a specific, isolated task, and the functionality of the app is emergent from the way these Views interact with each other.

Reed Log's Structure

While the views are a central part of SwiftUI, they're only a part of the whole app's structure. SwiftUI is simply the framework through which one creates and uses UI elements to display things on the screen. The general structure of the whole project that I have adopted, and that is most commonly used with SwiftUI is the MVVM design pattern.

MVVM

The MVVM (Model-View-ViewModel) design pattern is a pattern that, as the name suggests, separates the whole project into three distinctive file types: Views, View Models, and Models. Diagram 1 depicts the data flow between these three types of files in a project that uses the MVVM pattern. Views are the files that house the stacked View structs that make up the UI. They have no direct access to the Model but have access to the View Model (VM). The View's purpose is to display data from the VM, facilitate user inputs, and pass the changes into the VM. On the other hand, the Model files only concern themselves with the data present in the project. The Model files have no direct access to View files, and have minimal access to the VM, apart from initial data retrieval. It exists to define the form of the data within the application and stores the data necessary using Core Data.



(Diagram 1): The diagram of the MVVM design pattern

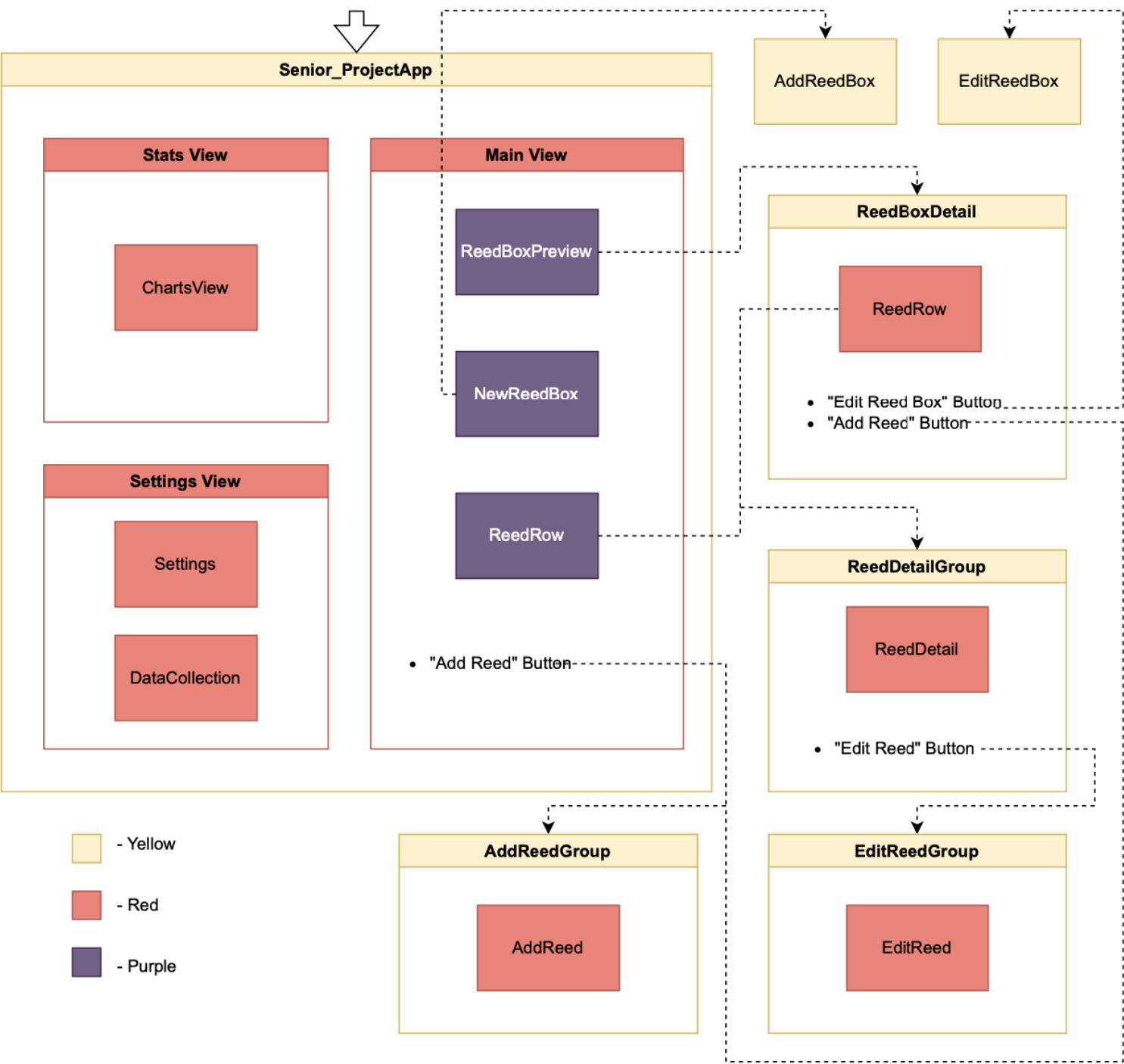
Lastly, the ViewModel is the interfacing agent between these two goals. It has direct access to both the View and Model files, and it is tasked with the processing of data. It is tasked with retrieving data from the Model, and processing it for usage in the View, as well as with

retrieving input data from the View and updating the data saved in the Model. This approach successfully separates any of the UI elements from the data itself, ensuring that UI elements can't directly access the data present within the Model. I will go through each of these groups in detail and show my implementation at these three levels of abstraction.

Views

As SwiftUI necessitates, the relationships between the View files that I have created is rather complex. The structure of these connections is modeled in Diagram 1, although a few convenience Views have been omitted for brevity. I have decided to split all Views into three distinct groups based on their role in the hierarchy: the yellow, red, and purple groups. The yellow group consists of views that are not hosted by another view — they are standalone screens that are not building blocks of other Views. On the other hand, the views within the red group are components of other views that host them. They can still host other views within them, but they are never displayed as standalone views. Lastly, the purple group consists of views that only exist to be a link to a view outside of the current view hierarchy. The purple views should be thought of in the same way as the buttons in the diagram, with said connections between views being shown as dashed lines.

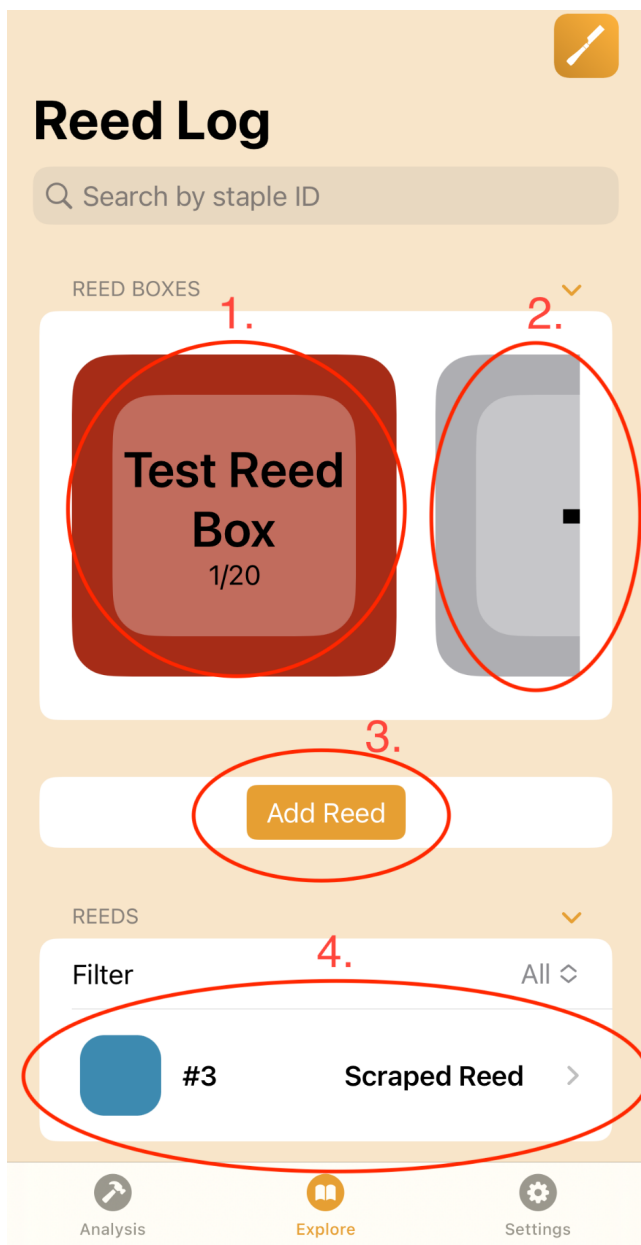
When entering the app, the user will be placed within the *Senior_ProjectApp* struct. Since this is the first view displayed, this View is not part of any other View, so the *Senior_ProjectApp* is labeled as yellow on Diagram 2. Said View directly hosts three other views within its *TabView* component: *MainView*, *SettingsView*, and *StatsView*. *TabView*, as the name suggests, hosts all of the screens within it as switchable tabs. In this case, the screens that are being hosted can be accessed through the icons at the bottom of the screen, with the Analysis tag corresponding to



(Diagram 2): The diagram of the main Views present in the app.

the StatsView, Explore tag to the MainView, and the Settings tag to the SettingsView. All three of these views host other views within them, but because they are parts of the Senior_ProjectApp

View, they are labeled as red. As demonstrated in the diagram, *MainView* is the only view within



(Screenshot 1): Explore Screen of the App

Views as Code

Let us see how these concepts are written out in code. Code Example 1 is the code for the *Senior_ProjectApp* file, while Code Example 2 comes from the *AnalysisView* file.

the *TabView* with links to other views

As pictured in Screenshot 1, this is what the *Senior_ProjectApp* looks like when the Explore tag is selected. The diagram dictates that within the *MainView*, there are three Views from the purple group and one “Add Reed” button. *ReedBoxPreview* from Diagram 2 is circled with the number 1, *NewReedbox* with the number 2, the “Add Reed” button with the number 3, and *ReedRow* with the number 4. As stated before, each one of these views fills the purpose of a button that leads to a separate View outside of the current View. For example, when the user clicks on the *ReedRow* view, one will be moved into *ReedDetailGroup*, changing the root View present.

```

12 @main
13 struct Senior_ProjectApp: App {
14     let minDragTranslationForSwipe: CGFloat = 50
15     @State private var selection = 2
16
17     var body: some Scene {
18         WindowGroup {
19             TabView(selection: $selection){
20                 let viewContext = DataController.shared.container.viewContext
21
22                 AnalysisView(rvm: ReedListViewModel(context: viewContext))
23                     .tabItem{
24                         Label("Analysis", systemImage: "hammer.circle.fill")
25                     }
26                     .tag(1)
27
28                 MainView(rvm: ReedListViewModel(context: viewContext), rbvm:
29                     ReedBoxListViewModel(context: viewContext))
30                     .environment(\.managedObjectContext, viewContext)
31                     .tabItem{
32                         Label("Explore", systemImage: "book.circle")
33                     }
34                     .tag(2)
35
36                 SettingsView(rvm: ReedListViewModel(context: viewContext))
37                     .tabItem{
38                         Label("Settings", systemImage: "gearshape.circle.fill" )
39                     }
40                     .tag(3)
41             }
42         }
43     }
44 }

```

(Code Example 1): Complete code of *Senior_ProjectApp*

As seen in line 13 of the Code Example 1 section, *Senior_ProjectApp* extends the *App* protocol, while the *AnalysisView* extends the *View* protocol, as seen in line 12 of Code Example 2. In contrast to the *View* protocol, which is used by every other *View* present in my application, the *App* protocol defines the entryway to the application. The more nuanced functionalities of the *App* and *Scene* protocols were not fully utilized within this project, so I will not talk much about them here.

```

12 struct AnalysisView: View {
13     @ObservedObject var reedListVM: ReedListViewModel
14
15     init(rvm: ReedListViewModel){
16         self.reedListVM = rvm
17     }
18
19     var body: some View {
20         NavigationView{
21             List {
22                 ChartsInStatsView(vm: reedListVM)
23             }
24             .scrollContentBackground(.hidden)
25             .background(Color("AccentColor2").edgesIgnoringSafeArea(.all))
26             .navigationTitle("Analysis")
27             .toolbar(){
28                 ToolbarItem{
29                     Button(){} label:{
30                         VStack{
31                             Image("Icon")
32                         }
33                     }
34                 }
35             }
36         }
37     }
38 }

```

(Code Example 2): Complete code of AnalysisView

Views in Practice

These code sections show how a view is defined and used as a component of other views. As seen in Code Example 2, the *AnalysisView* is defined as a struct with a set of variables stored within. The most crucial variable within a View struct is the `body` variable, which stores all visual information about UI placement. It defines the relationships between the components from which the newly-created View emerges. Other variables within the View can fill many different roles, often defined by property wrappers placed before the variable definition. The two examples of property wrappers present in these code examples are `@State`, and `@ObservedObject` wrappers.

Property Wrappers

Apple Documentation defines the `@State` wrapper as “A property wrapper type that can read and write a value managed by SwiftUI” (Apple Documentation, *State*), and variables with said wrapper will be used as short-term data storage for user input. The wrapper marks variables that will change during a View’s lifecycle, triggering Views dependent on the said variable to re-render. This functionality allows one’s input to be immediately seen within the view, changing all data displayed using the said variable. For example, the `selection` variable within the *Senior_ProjectApp* View stores which screen the user is currently on, allowing the TabView interface to dynamically switch between views based on the variable based on user input. If `selection` did not have the `@State` wrapper, the screens would not change until the user would leave the screen, forcing a new lifecycle on the View. Since the `@State` wrapper is applied to variables within Views, it will be relegated to controlling UI logic only, as shown in the example above.

On the other hand, the `@ObservedObject` wrapper is defined as “A property wrapper type that subscribes to an observable object and invalidates a view whenever the observable object changes.” (Apple Documentation, *ObservableObject*) In many ways, it is very similar to the `@State` wrapper, but it is “now using an external reference type rather than a simple local property like a string or an integer.” (Hudson, 2021f) In other words, the `@ObservedObject` wrapper serves the same purpose as the `@State` wrapper, only used with instances of more complex structures and their internal states. It is a wrapper that needs to be applied to an instance of an object that extends the *ObservableObject* protocol, and it will be applied to every single instance of a VM within the View. Since the purpose of VMs is to process data for usage within the View, the Views need to be kept up to date with the VMs, and the `@ObservedObject`

wrapper is ideal for that purpose. The exact way that said wrappers work will be discussed at length in the View Models section.

Using Views and Method Calls

Just like other components used throughout the creation of a View, the Views themselves are used by simply referencing them as if they are functions. As seen on line 22 of Code Example 1, the `AnalysisView` is used within the body of the struct, with its' necessary parameters present within parentheses. On top of parameters, SwiftUI allows developers to customize these views using method calls. The View protocol provides structs with several methods that modify the way the View is being displayed. These methods are present in both Code Examples, although we will focus on Code Example 2.

Just like in Objective-C, Swift and SwiftUI use the dot syntax to access these methods, as seen in lines 24-34. The interesting difference between a standard use of the dot syntax is that within SwiftUI's Views, these calls are all stacked on top of each other. Within Code Example 2, the List View, which houses the `ChartsInStatsView()`, has four different function calls attached. The first two change the way the List looks, with `.scrollContentBackground()` setting the standard background of a pre-made List to be invisible and setting the actual background's color to "AccentColor2" with `.background()`. The other two functions have to do with the outer NavigationView in which the List View is housed, with the text at the top of this screen being set by `.navigationTitle()` and adding the app's logo in the corner within the toolbar by `.toolbar()`.

ViewModels Within Views

As discussed in the MVVM section, the Views have no direct access to the app's data. The only way for them to access the data is through the usage of View Models. For now, let's treat these VMs as black boxes that simply have all of the data necessary for the View. An instance of one of these VMs can be found on line 13 of Code Example 2 under the name `ReedListViewModel`. Within this Code Example, this VM is simply passed along to the `ChartsInStatsView()`, so to understand how the VM is actually used, let us look at Code Example 3.

```

11 struct ChartsInStatsView: View {
12     @ObservedObject var vm: ReedListViewModel
13
14     var body: some View {
15         Section("Reed Box"){
16             HStack{
17                 Text("Total Reeds: ")
18                     .bold()
19
20                 Text(String(vm.num))
21             }
22             .centerHorizontally()
23             HStack{
24                 Text(String(self.percent(vm.numBlanks, vm.num)) + "% Blanks")
25                     .bold()
26                 Spacer()
27                 Text(String(vm.numBlanks))
28             }
29             HStack{
30                 Text(String(self.percent(vm.numScraped, vm.num)) + "% Scraped")
31                     .bold()
32                 Spacer()
33                 Text(String(vm.numScraped))
34             }
35             HStack{
36                 Text(String(self.percent(vm.numInUse, vm.num)) + "% In Use")
37                     .bold()
38                 Spacer()
39                 Text(String(vm.numInUse))
40             }

```

(Code Example 3): Parts of the ChartsInStatsView code

Code Example 3 shows parts of the `CharsInStatsView`, which houses only two variables: the View Model and the body of the View. The `vm` variable has the `@ObservedObject` property wrapper, which means that if data within the VM changes, any of the Views using said VM will update to reflect that. This VM gives the view access to the reeds present within the app's storage and a number of statistics about the reeds present. To access the data within the VM, `ChartsInStatsView()` simply references its variables through the dot syntax, as seen throughout Code Example 3.

On top of using VMs to display data on the screen, VMs also take in user inputs from the Views. This process is done in a similar way, with the variables being referenced through the dot notation. One critical difference in usage between displaying and changing VM's data is the necessity to reference the variable binding instead of just its value. Swift handles that through the use of the `$` symbol, which works similarly to a pointer in other C-based languages. In other words, "Swift property wrappers use (the `$` symbol) to provide two-way bindings to their data." (Hudson, 2022)

```

104         Section(header: Text("Tying Information")){
105             TextField("Staple Type", text: $vm.stapleType)
106             TextField("Staple Number", text: $vm.stapleID)
107             TextField("Tie Length", text: $vm.tieLength)
108                 .keyboardType(.numberPad)
109             ColorPicker("Thread Color", selection: $vm.threadColor)
110         }

```

(Code Example 4): Parts of the AddReed View code

Code Example 4 is a section of the code from the AddReed View. The section shown manages the process of collecting reed data on the tying process, and it is done through the usage of `TextField()`, or a simple View component that hosts a container for user-inputted `text`. It

takes in a label parameter displayed when no information is entered, and a `text` parameter, which is a binding to the variable used to store the data collected by the `TextField()`.

View Models

As seen throughout the previous section, the View Models are a critical part of the whole application, linking the UI's logic from the Views to the actual data stored in the Model. The VMs need to extend the `ObservableObject` protocol so that they can be used by the Views to dynamically display information and collect the information from the Views themselves. Beyond processing the data in any way the app necessitates, the VMs also have to be able to access the data stored in the Model and save said data into the Model when needed.

VMs and Views

To see how said functionality is implemented within code, let us look at the VM used within `ChartsInStatsView`, mainly the `ReedListViewModel`. This specific VM serves two distinct functions: it is the VM that concerns itself with a list of all reeds present within the application. It also handles the retrieval of data from the `NSManagedObjectContext`.

As seen in Code Example 5.1, the VM is now a class rather than a struct, and it extends three protocols: the `NSObject`, `ObservableObject`, and `NSFetchedResultsControllerDelegate` protocol. The first protocol is a simple prerequisite for the second one and will not be discussed in this paper. The VM does not extend the View protocol, and it acts like a regular class, oblivious to the presence of a UI. It has several variables with `@Published` property wrappers, and it has a mysterious context variable of the type `NSManagedObjectContext`. Most importantly, it has a variable called `reeds`, which

```

12 // The View Model class for the list of Reeds, handling fetches, and updating lists. Stores the
13 // non-core data list of reeds.
14 class ReedListViewModel: NSObject, ObservableObject, NSFetchedResultsControllerDelegate{
15     private (set) var context: NSManagedObjectContext
16
17     // Array of @Published Reed objects that's accessible by the view
18     @Published var reeds = [Reed]()
19
20     @Published var num: Int
21     @Published var numBlanks: Int
22     @Published var numScraped: Int
23     @Published var numInUse: Int
24     @Published var numDestroyed: Int
25     @Published var averageSuccess: Float
26     @Published var averageLoudness: Float
27     @Published var averagePitch: Float
28
29     @Published var avLeftL: Double
30     @Published var avLeftM: Double
31     @Published var avLeftR: Double
32     @Published var avRightL: Double
33     @Published var avRightM: Double
34     @Published var avRightR: Double
35     @Published var avBottomLeft: Double
36     @Published var avBottomRight: Double

```

(Code Example 5.1): The Beginning of ReedListViewModel Class

houses all reeds retrieved from the Core Data database. Other critical parts of this VM include its `init()` function, shown in Code Example 5.2, and a `controllerDidChangeContent()` function, shown in Code Example 5.3.

Property Wrappers Revisited

First, let's discuss the parts of this VM that directly correspond to the functionality described in the View section. The Views were designed in such a way as to rely on the VM to provide them with dynamically-updated data to display, as well as storage for the data collected by the views. As discussed previously, this is done through the `@ObservedObject` property wrapper that was applied to an instance of an object extending the `ObservableObject` protocol. One key aspect of this relationship that was left hidden within the VMs was the `@Published` property wrapper. It defines which properties should be looked at when determining whether a

class changed its state. The VM extends the ObservableObject protocol to ensure that it can be “observed” by a View, and labels its internal variables that will have to be updated dynamically through the `@Published` property wrapper. The View then “observes” the VM by creating an instance of it with an `@ObservedObject` property wrapper attached, and updates all internal views that rely on the VM. The `@Published` property wrapper also allows the view to retrieve the binding of the variable to allow the View to pass user input along to the VM. Because of this critical functionality, all variables accessible by the View within the VMs will have the `@Published` wrapper attached.

View Models and the Model

Beyond interfacing with the Views, the VM must also have functionality associated with interfacing with the Model. As described in the MVVM section, the Model’s functionality is to define the form of the data present with the application, through the creation of types like `Reed` or `ReedBox` seen throughout the app. Beyond defining the data, the Model also concerns itself with managing the databases in which raw data is actually stored. The VM layer then needs to access the information retrieved in the Model, and process it.

Both the context variable as well as the `NSFetchedResultsControllerDelegate` protocol work in tandem to allow the VM access to the data within the databases managed by the Model. While the nature of the `NSManagedObjectContext` object will be explored in more detail within the Model section, one needs to understand the general functionality of the context to understand how it facilitates said interaction.

Context Introduction

While the functionality of the context variable is heavily dependent on a number of other components within the Model, its functionality can be described in practice. In general terms, the context, or an object of type `NSManagedObjectContext`, is “an environment where (one) can manipulate Core Data objects entirely in RAM.” (Hudson, 2021a) In other words, it is a container that houses the data retrieved from Core Data that facilitates the interaction between the data stored within the database, and the VM that processes said data.

Within my application, only one context is created, and it is passed to every single VM that needs to display or modify any data within the program. This can be seen throughout all of the code examples provided, with one of the parameters necessary for the creation of a VM always being a context object. The context is retrieved from the Model on line 20 of Code Example 1 at the inception of the application. The details of what exactly happens within the `DataController` file will be described within the Model section.

As described above the context houses all of the data retrieved from Core Data, but how does the program actually use this data? Sadly, it is not as simple as accessing some sort of internal variable within the context. Instead, it requires the developer to use a completely separate set of tools to manage that interaction.

NSFetchedResultsControllerDelegate

The `NSFetchedResultsControllerDelegate` protocol concerns itself with this specific interaction, and `ReedListViewModel` extends said protocol to gain the ability to retrieve data from the context. The protocol allows for the VM to “fetch” data from the context through the use of `NSFetchedResultsController` and `NSFetchRequest` objects. These objects facilitate the continuous retrieval of data from the context, dynamically updating data stored

within the `reed` variable when the context is modified by using the `controllerDidChangeContent()` method.

The `NSFetchedResultsController` provides a way to dynamically retrieve data from the context, with the ability to filter said data based on a number of predicates housed within the `NSFetchRequest` object. In other words, `NSFetchRequest` “is the class that performs a query on your data, and returns a list of objects that match.” (Hudson, 2021b) These classes allow the application to only work with data that’s immediately needed, removing the amount of entities stored within the data structures of the app. Since my app functions as a log for a single person to use, it will never have to deal with any data exceeding 1000 in number of entries. The performance of storing reeds in said order of magnitude is later tested, and my approach works very well up to three orders of magnitude, perfectly fulfilling the role it was designed for. Since the app also needed to have access to the entire collection of reeds at once to perform analysis, I decided to limit my usage of the predicates to simply retrieving all reed entities into internal storage.

```

40     init(context: NSManagedObjectContext, fetchRequest: NSFFetchRequest<Reed>){
41         self.context = context
42
43         // @Published properties initialized with standard values
44
45         reedFetchedResultsController = NSFetchedResultsController(fetchRequest: fetchRequest,
46             managedObjectContext: context, sectionNameKeyPath: nil, cacheName: nil)
47         super.init()
48         reedFetchedResultsController.delegate = self
49         do{
50             try reedFetchedResultsController.performFetch()
51             guard let reeds = reedFetchedResultsController.fetchedObjects else {
52                 return
53             }
54             self.reeds = reeds
55             updateStatValues()
56         } catch {
57             print(error)
58         }
59     }

```

(Code Example 5.2) : `init()` function from `ReedListViewModel`

Code Example 5.2 shows how these concepts are actually implemented. Within the `ReedListViewModel` class's `init()` function, an instance of `NSFetchedResultsController()` is created on line 45 under the name `reedFetchedResultsController`, with instances of `fetchRequest` and `context` as its parameters. The key parts of this code section though are lines 48 through 58. Upon the creation of `ReedListViewModel`, the `reedFetchedResultsController` calls its method `.performFetch()` on line 49, which retrieves all of the data from the context that satisfy the predicates defined within `fetchRequest`. Said `fetchRequest` within my application will simply fetch all reeds present within the application, excluding one convenience reed that serves as a template for other reeds. The result of the fetch is then retrieved from the `reedFetchedResultsController` on line 50, and stored within the VM's array on line 54.

ControllerDidChangeContent()

To use the `NSFetchedResultsController` object, the object needs to be housed within a class that extends the `NSFetchedResultsControllerDelegate` protocol. The protocol also necessitates a definition of the `controllerDidChangeContent()` method, which lets the developer specify the behavior when the context changes. This functionality gives the developer the freedom to create properties that are computed at the moment the data changes, allowing for an easy implementation of any statistical analysis of the reeds' properties

Code Example 5.3 shows the definition of `controllerDidChangeContent()` within `ReedListViewModel`. The function simply re-accesses the result of the fetch done in the Code Example 5.2, and re-assigns it to the VM's reeds array. In other words, one could once again

```

213 // Runs when something in the Core Data storage changes, and updates the reed array for the view
214 func controllerDidChangeContent(_ controller: NSFetchedResultsController<NSFetchRequestResult>) {
215     guard let reeds = controller.fetchedObjects as? [Reed] else {
216         return
217     }
218
219     self.reeds = reeds
220     updateStatValues()
221 }

```

(Code Example 5.3): controllerDidChangeContent() definition from ReedListViewModel

think of NSFetchedResultsController as a black box that manages all of the data within the database, and simply returns the most recent set of entities that fit the description provided by fetchRequest.

Because this method is called every time any of the data within the internal database changes, it is the perfect time to recalculate all of the variables shown in Code Example 5.1. Since all of these variables are averages of certain parameters from reeds present in the application that will be directly displayed on screen in StatsView, they need to be updated any time anything changes within the app. This is accomplished in both Code Examples 5.2 and 5.3 through the usage of `updateStatValues()`, which performs simple calculations and updates based on the set of reeds present.

Storing New Data

So far, most of the VM's functionality centered around dynamically responding to changes in the data stored within the Model, but the VM also needs to have the ability to update the Model's data. Since the View has no direct access to the Model and vice versa, the VM is the layer that also needs to handle said interaction. Thankfully, the `NSManagedObjectContext` file provides said functionality, allowing the VM to not concern itself with how exactly the data is

being saved. The context is simply notified that it should attempt to save, which it handles internally.

Code Example 6 shows a simplified version of the save function from `AddReedViewModel`. The MV extends the `ObservableObject` protocol, and has a number of variables with `@Published` property wrappers that receive information from the `AddReedView`. When the “save reed” button is pressed, the following function gets called. The function

```
181     func save() -> Bool{
182         let newReed = Reed(context: context)
183
184         // Parameter assignments of the Reed object
185
186         try? context.save()
187         return true
188     }
```

(Code Example 6): save() function from AddReedViewModel

creates a new `Reed` object and passes the context into it on line 181. This step effectively creates a new instance of an entity provided by the Model within the context provided, letting Core Data know that it needs to manage it as well. The `Reed` object is then filled with the data provided by the user, and the modified context is saved on line 186. As seen in the example above, most of the work in saving new data is done within the model, allowing the VM to put its focus elsewhere.

Model

As explained within the View Model chapter, the job of the Model is to define the form that the data will take by providing the developer with custom object types that store desired

information. The Model also acts as a database management system, dynamically retrieving data from the Core Data databases, and updating said databases with new data when prompted. Both of these functionalities will be explored in this section in detail.

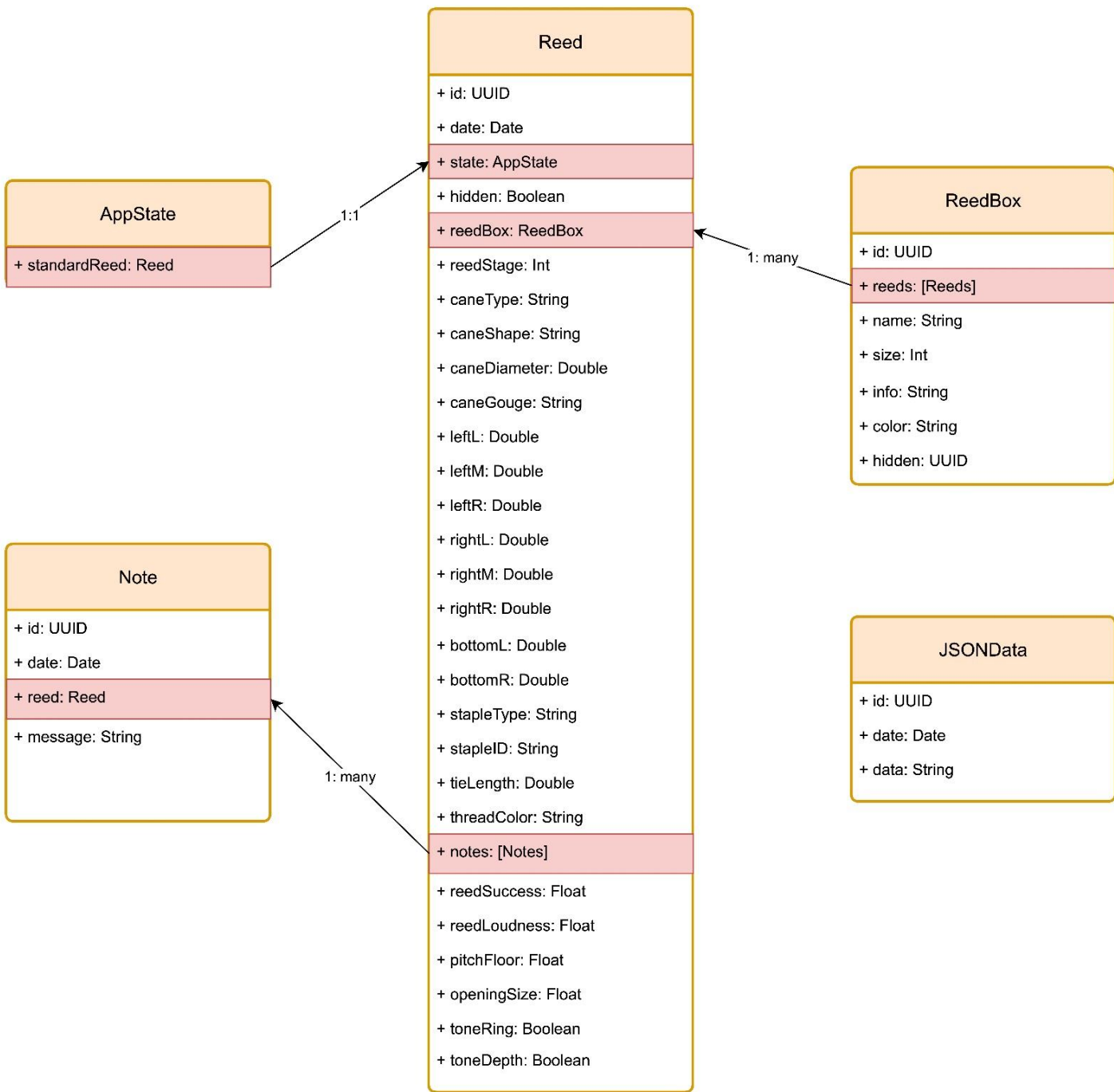
Core Data Revisited

The tool that helps the Model complete all of its responsibilities is Core Data. It is defined as “Apple's object graph and persistence framework, (and) it reads, writes and queries collections of related objects while also being able to save them to disk.” (Hudson, 2021c) It both creates a Core Data model which defines entities that get used as data types within our program, and automatically manages them. It provides a way for the developer to interact with the databases that store one’s data without needing to manage said interactions by hand.

Entities and Data Types

When first starting one’s work with Core Data, one first needs to create a Core Data model. Said model defines all of the entities present within the program, as well as the relationships between said entities. Diagram 3 shows all of the entities present within my program.

Within Diagram 3, each box corresponds to a different entity present within my Model, with each row inside of the box representing a single data point being saved within it. Each one of these internal attributes is added to an entity through a visual editor within XCode, and each attribute needs to be chosen from a pre-set amount of data types. Said list of data types does not include any of the custom data types created within Core Data, meaning that no entity could ever contain other entities as one of its attributes. Instead, Core Data allows one to specify a set of relationships between entities, which satisfy the same purpose. These entities let Core Data know



(Diagram 3): Relationship between Entities within the Core Data model

that each instance of a specific entity has the ability to contain instances of other entities, with the number of entities stored being specified by the type of relationship. These relationships can either be one-to-one, or one-to-many, and have to have a source and destination.

When a source entity has an one-to-many relationship with a destination entity, the source entity has the ability to store multiple destination entities, while each destination entity can only have one source entity. Conversely a one-to-one relationship is characterized by the source entity having one destination entity, and vice versa. The labels also underline the conceptual idea that certain entities contain other entities, but both can still access each other regardless of whether they are the source or destination in a relationship.

As seen in Diagram 3, there are five entities present within my program: Reed, ReedBox, Note, JSONData, and AppState. Since the purpose of this application is to store data about reeds, it should not be too surprising that our Reed entity would be the one with the most complexity. The Reed entity houses twenty six separate parameters, and has relationships to three other entities.

Within my app, each reed needs to be part of a single reed box, and house a number of notes within it. Conversely, each reed box needs the ability to house a number of reeds within it, while each note can only belong to a single reed. When translated into Core Data entities, the Reedbox entity has an one-to-many relationship with the Reed entity, and the Reed entity has an one-to-many relationship with the Note entity. The entities on the other sides of the relationships are accessible through attributes highlighted in red within Diagram 3.

Both AppState and JSONData are convenience entities with limited functionality; the AppState entity stores a single Reed entity that is used to create a set of standard reed values, and

the JSONData entity stores a saved version of all of the reed data from a specific date, allowing the user to get access to multiple versions of one's storage.

CoreDataClass and CoreDataProperties

While Core Data entity definition can be done completely through the visual editor, getting access to the actual class definition of said entities gives the developer a lot more freedom in the development of their app. These class definitions are housed in a set of two files per entity; each entity has a CoreDataClass and CoreDataProperties class. The first is a straightforward definition of the data type as a class, while the second extends said class to add internal functionality to these entities.

A detailed description of how these classes work is not necessary to understand how my code works, so its functionality will be only explained briefly. The extension accomplishes three main roles: it programmatically defines the entities' attributes as variables with a `@NSManaged` property wrapper attached, and defines relationships as variables or `NSSet`s, depending on the type of the relationship present. The extension also creates an instance of a `NSFetchRequest` object through the use of the `createFetchRequest()` method. Whenever a `NSFetchRequest` object is used throughout the Model and VM layers, it is created through the calling of that method on the desired data type.

DataController Class and NSPersistentContainer

Now that we know the relationship between these entities, as well as how they are actually implemented within the code, let us look at the DataController class, which handles interfacing with the actual Core Data database. The code is very short and simple, and it relies on the `NSPersistentContainer` class to accomplish all of its goals. Apple Documentation

defines `NSPersistentContainer` as a “container that encapsulates the Core Data stack in your app”, which simplifies “the creation and management of the Core Data stack by handling the creation of the `NSManagedObjectModel`, `NSPersistentStoreCoordinator`, and the `NSManagedObjectContext`,” (Documentation `NSPersistent`) `NSManagedObjectModel` being the Core Data model created in the previous section, and the `NSManagedObjectContext` being the context passed around between Views, and VMs. Beyond the `NSPersistentStoreCoordinator`, which is not used within my project, both model and context have been explained in detail in the previous section. The only thing left to discuss is the creation of the container itself.

```

11 class DataController: ObservableObject{
12     let container = NSPersistentContainer(name: "ReedModel")
13     static let shared = DataController()
14
15     init(){
16         container.loadPersistentStores { description, error in
17             if let error = error{
18                 print("Core Data failed to load: \(error.localizedDescription)")
19             }
20         }
21     }
22 }

```

(Code Example 7): Parts of the DataController class

Code Example 7 shows the definition of `DataController` class. It creates an instance of itself within a static variable called `shared`, and stores the `NSPersistentContainer`. Within the `init()` method, the `loadPersistentStores()` method is called, which either loads pre-existing databases, or creates new ones, completing the implementation of Core Data within the application.

App Evaluation

The use of the MVVM design pattern is the principal reason for this app's success. The app is structured around separating all of the code concerned with UI logic from the code that actually works with the data of the project, which made the project efficient all around. The resulting code's functionality was clear, and never fell into any of the major coding pitfalls. None of the UI elements ever interfered when I was implementing the Core Data or the actual functionality of the application, and debugging everything within the app was painless and effortless.

Beyond code clarity, I wanted to see what the actual performance of the application was when using the MVVM pattern with SwiftUI. I decided to test three metrics: the size of the app, the speed of adding reeds and converting them into JSON, and the size of each reed when saving it within Core Data. The first metric was very simple to check, since when publishing an app on the App Store, one has access to the size of the whole application. The app ended up being extremely small, occupying around 2MB of space when downloaded.

Speed Tests

To test the speed at which reeds are added, I wrote a test function `testReedStorage()`, which creates a specified amount of reeds with random data within it. The function also times the process from the creation, to the saving of the reed in Core Data's database. The implementation of said function can be seen in Code Example 8, which is housed within the `AddReedViewModel` file. To get the current time within the code's execution, `CFAbsoluteTimeGetCurrent()` was used as seen in lines 209 and 244, and the difference between recorded times was displayed in the logs of the application. The exact same method was

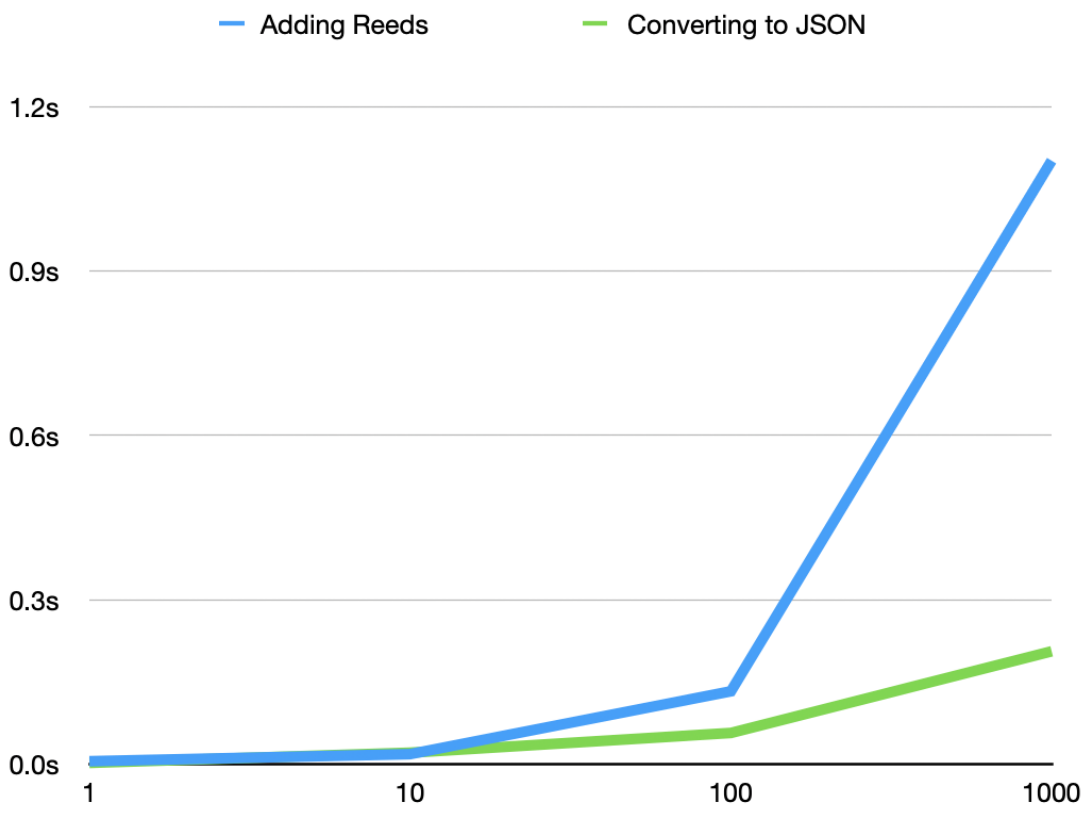
used when testing JSON conversion. Said speed test was done on a native emulator of iPhone 14 Pro on a M1 MacBook Pro.

```
209     func testReedStorage(numReeds: Int){
210         let start = CFAbsoluteTimeGetCurrent()
211         for _ in 1...numReeds{
212             let newReed = Reed(context: context)
213
214             // Setting randomly generated Reed properties
215
216             try? context.save()
217         }
218         let diff = CFAbsoluteTimeGetCurrent() - start
219         print("Adding of \(numReeds) took \(diff) seconds")
220     }
221 }
```

(Code Example 8): testReedStorage() function from AddReedViewModel

Results

Graph 1 shows the results of said tests. The speed of addition increased linearly proportional to the increase of the number of reeds being added. This was expected, because the Core Data database was only accessed at the end of the process; whenever a new reed was being created, only the context was modified. The context was then saved at the end of the process, making sure that the most difficult work is only done once. The MVVM structure of my project guaranteed this type of approach, with all of the saving process being done by the `NSPersistentContainer` and the `NSManagedObjectContext` created within the `DataController` file.



# of Reeds	Adding Reeds	Converting to JSON
1	0.0052419900894165	0.00220108032226563
10	0.0182429552078247	0.0208430290222168
100	0.13290798664093	0.0568729639053345
1000	1.10145401954651	0.20592999458313

(Graph 1): Results of the speed test

Memory Tests

The last metric that I decided to test was the actual size of each Reed object being stored within Core Data. Sadly, because of the sheer amount of abstraction present within the implementation of Core Data, this process was not as simple as the tests done beforehand. For one, Core Data actually manages all of the stored data between multiple different SQLite. There is a main database that eventually stores all of the data present within the app, and the WAL

(Write-Ahead Logging) database, which provides a faster, less permanent alternative for the data to be stored temporarily (Lee, 2020). The WAL allows Core Data to prevent “data loss by writing new transactions in an in-between journal file”, without the need for “keeping a copy of the original unchanged database content in the journal file” (Lee, 2020). It allows for faster, and more efficient logging, which sadly makes the test a lot harder to complete. Beyond this problem, Core Data dynamically manages the memory allocation of these databases, making the process of testing extremely difficult and confusing.

To test the actual size of each Reed entity, I decided to approximate the size based on the changes within these two databases. For this purpose I wrote the `testCoreDataSize()` function, which can be seen in Code Example 9. The function accesses the path to the actual SQLite files used within Core Data, and prints out the size of said files using the function `getSqliteStoreSize()`. To test a number of different data points, I filled the application with a specified number of reeds spanning from 0 to 5000, and used the test function.

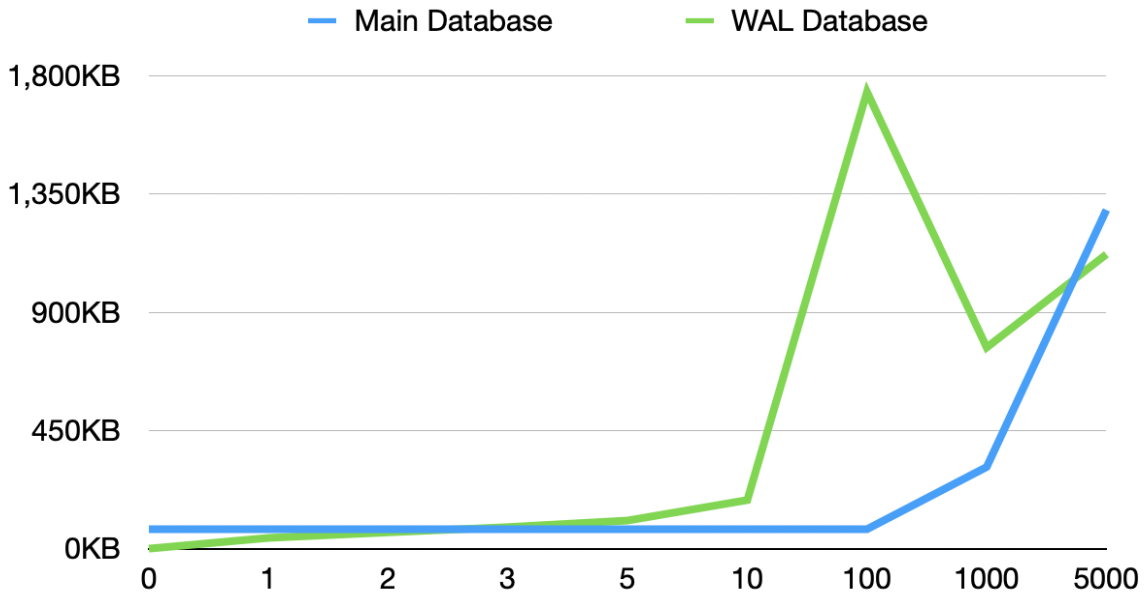
```

40     func testCoreDataSize(){
41         // Get the URL of the SQLite databases
42         guard let storeUrl =
43             self.container.persistentStoreCoordinator.persistentStores.first?.url
44             else {
45                 print("There is no store url")
46                 return
47             }
48         let newFileName = storeUrl.absoluteString.appending("\("-wal")")
49
50         // Show the size of said databases
51         print("The size of the store is:
52             \(\self.getSqliteStoreSize(forPersistentContainerUrl: storeUrl))")
53         print("The size of the wal is:
54             \(\self.getSqliteStoreSize(forPersistentContainerUrl: URL(string:
55             newFileName!))")
56     }

```

(Code Example 9): testCoreDataSize() function from DataController

Results



# of Reeds	Main Database	WAL Database
0	74	0
1	74	41
2	74	62
3	74	82
5	74	107
10	74	185
100	74	1,739
1000	311	766
5000	1,290	1,121

(Graph 2): Results of the size test

Graph 2 demonstrates the results of this test. Please note that the horizontal axis within the graph is not linear, rather showing only the data points that were tested. When comparing the changes in storage between the databases, one can approximate one reed to take up under 20KB. This number is an approximation though, since every time a reed was added it would take up a different amount of space. Between 0 and 100 reeds, the WAL database was mainly utilized, taking up as much as 1739KB in the case of 100 reeds. The data then became managed between

both SQLite files in the case of 1000 and 5000 reeds, with the final breakup being somewhat evenly distributed.

As the graph suggests, when the WAL file receives a certain amount of changes, said changes are made permanent by transferring them to the main database. Between 100 and 1000 reeds stored, the data began being saved to the permanent storage, and suspect that if my application was working with a larger amount of data, more data would be immediately stored within the main database than the WAL.

Conclusion

As seen from the results of these three tests, my application is rather efficient. The app takes up minimal space, completes its functionality in negligible time, and stores necessary information with minimal memory usage. The process of adding reeds and converting reeds into JSON both take under 0.01 seconds, meaning that the application's speed will never be a problem under normal conditions. An issue only arose when adding over a 1000 reeds at once, an operation that one cannot complete outside of testing mode. The size of each data point does not pose an issue either, with each reed taking up under 20KB of space, which is dynamically allocated to proper databases by Core Data. Because of this, the size of said databases will never pose a problem to the user. Because of this great foundation, the application will stay efficient even as functionality is added in the future.

Future Projects

Although I made this application with personal use in mind, the app could be used as a study tool for data collection. I decided to implement a JSON converter for this specific reason. One could easily collect data from oboists by asking them to use the app, and periodically sending them the JSON file generated within the application.

Studies Involving the Reed Log

The data collected within the application can give a lot of insight into the relationships between variables within reed making. Said information can be analyzed either alone, or in tandem with a number of other measurements one can make. One could, for example, collect a large amount of data from a number of oboists from different educational backgrounds and analyze the differences between objective measurements of the reed. One then could find specific arrangements of the data that maximize success, and find out why these arrangements have said effect on the reed. One could explore and classify the specific ratios between gouge measurements, staple dimensions based on brand, and shape used to find the arrangements that work best for all of the tools available. One could define the relationships between the tools, allowing one to find the best tools to use given a set of measurements, or one could attempt to find markers within the early stages of the reed that foreshadow success or failure.

The data collected within the app could also be used together with other data collected in other ways. In a similar fashion to the 2013 study *A Study of Oboe Reeds*, one could collect the sound spectra of a reed's crow on top of all of the information collected through the application, and analyze the data looking for trends. Finding the connections between these spectra and the internal architecture of the reed would greatly increase our understanding of reed making.

Commercial Uses

Beyond academic use, the application is perfect for both educational and personal purposes. When used by a student, the application would be a perfect tool for systematizing one's reeds. It would allow each student to objectively look at their progress and spot issues that could not be easily spotted without analysis or a deep understanding of reeds. When used by a professional, it can immensely help to track one's reeds over long periods of time. While reeds do not last a long time, professional oboists often make reeds in large batches, leaving them for use at a later date. The application would help keep track of said reeds, and allow for professionals to emulate most successful arrangements while avoiding common issues.

The app is currently available on the App Store, but some changes would need to be made for it to be ready for general use. For one, the reliance on pre-made parameters to track adds an unnecessary amount of tedium to oboists that do not want to track all of the information that the app tracks right now. To convert this application for commercial use, I would make the parameters tracked by the application be editable, with each user creating their own model of what they wish to track about their reeds. I would also want to include a more comprehensive Notes section that can be applied to specific tools, as well as greatly expand the analysis section, allowing users to create experiments of their own creation.

The application could also include functionality that would suggest the use of specific tools based on the information entered. It could also analyze the actual scraping of the reed and test results, and suggest fixes based on all of the information entered. While the implementation of such functionality would be a lot more complicated, the idea would be very interesting to explore in future projects.

Conclusion

The art of reed making is extremely complex - the interplay between parameters needs to be extremely precise, and the process is extremely painstaking. Such requirements seem above human ability, yet people still manage to do it. Technology provides us with the ability to ease some of the burden, simplifying the process and making it more accessible, but it allows us to enhance the process as well.

The app attempts to help oboists keep their reeds systematized, improving efficiency of the process, as well as the quality of the final product. The app was made in Swift using the SwiftUI UI framework, and utilized the MVVM design pattern. SwiftUI allowed for the focus to be placed on the actual design of the UI, and MVVM in tandem with Swift itself provided with an efficient base for the application. Said efficiency was tested using a number of tests, which showed that both memory allocation and speed of the application were more than satisfactory.

This application is a proof of concept that I hope to further develop into a commercially-available application. The app as I imagined it would greatly benefit oboists of all varieties.

Bibliography

- Apple. (n.d.-a). *NSFetchedResultsControllerDelegate* [Documentation]. Apple Developer Documentation. Retrieved May 2, 2023, from <https://developer.apple.com/documentation/coredata/nsfetchedresultscontrollerdelegate>
- Apple. (n.d.-b). *NSPersistentContainer* [Documentation]. Apple Developer Documentation. Retrieved May 2, 2023, from <https://developer.apple.com/documentation/coredata/nspersistentcontainer>
- Apple. (n.d.-c). *ObservableObject* [Documentation]. Apple Developer Documentation. Retrieved May 2, 2023, from <https://developer.apple.com/documentation/combine/observableobject>
- Apple. (n.d.-d). *State* [Documentation]. Apple Developer Documentation. Retrieved May 2, 2023, from <https://developer.apple.com/documentation/swiftui/state>
- Gjebic, J. (2013). A Study of Oboe Reeds. *Grand Valley State University*, 22.
- Hudson, P. (2021a). *Adding Core Data to our project: NSPersistentContainer - a free Hacking with Swift tutorial*. Hacking with Swift. <https://www.hackingwithswift.com/read/38/3/adding-core-data-to-our-project-nspersistentcontainer>
- Hudson, P. (2021b). *Loading Core Data objects using NSFetchedRequest and NSSortDescriptor—A free Hacking with Swift tutorial*. Hacking with Swift. <https://www.hackingwithswift.com/read/38/5/loading-core-data-objects-using-nsfetchedrequest-and-nssortdescriptor>

Hudson, P. (2021c). *Setting up—A free Hacking with Swift tutorial*. Hacking with Swift.

<https://www.hackingwithswift.com/read/38/1/setting-up>

Hudson, P. (2021d, February 9). *SwiftUI vs Interface Builder and storyboards—A free*

SwiftUI by Example tutorial. Hacking with Swift.

<https://www.hackingwithswift.com/quick-start/swiftui/swiftui-vs-interface-builder-and-storyboards>

Hudson, P. (2021e, February 9). *What is SwiftUI? - A free SwiftUI by Example tutorial*.

Hacking with Swift.

<https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui>

Hudson, P. (2021f, September 3). *What's the difference between @ObservedObject, @State, and @EnvironmentObject? - A free SwiftUI by Example tutorial*. Hacking with Swift.

<https://www.hackingwithswift.com/quick-start/swiftui/whats-the-difference-between-observedobject-state-and-environmentobject>

Hudson, P. (2022, December 1). *Two-way bindings in SwiftUI - a free SwiftUI by Example tutorial*. Hacking with Swift.

<https://www.hackingwithswift.com/quick-start/swiftui/two-way-bindings-in-swiftui>

Lee, A. van der. (2020, August 25). Write-Ahead Logging (WAL) disabled to force commits in Core Data. *SwiftLee*. <https://www.avanderlee.com/swift/write-ahead-logging-wal/>

Mejia, R. (2019, December 30). Declarative and Imperative Programming using SwiftUI and UIKit. *Medium*.

<https://medium.com/@rmeji1/declarative-and-imperative-programming-using-swiftui-and-UIKit-c91f1f104252>

Salter, G. (2018). *Understanding the Oboe Reed*. Bearsden Music.

Waddell, G., & Williamon, A. (2019). Technology Use and Attitudes in Music Learning.

Frontiers in ICT, 6. <https://www.frontiersin.org/articles/10.3389/fict.2019.00011>