Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Tree

time management app

Supervisor:
Zoltán Gera
Lecturer

Author:
Csaba Dabis
Computer Science B.Sc.

Budapest, 2021

Table of contents

1. Acknowledgements

I would like to thank Zoltán Gera, Gábor Horváth, Zoltán Porkoláb and Melinda Tóth for mentoring me in my compiler engineering research projects. Without Richárd Szalay's, Kristóf Umann's and Norbert Czakó's notes and tips the university would have been a lot more difficult to finish.

I also would like to express my gratitude for the European Union scholarship programs and the Google Summer of Code stipends which made this thesis possible.

Special thanks to Zoltán Gera, Artem Dergachev, Sarah Rogers, Denise McCarney, Mon-Ping Wang, and Patrick Shyu to guide me in my programming career.

2. Introduction

2.1. Overview

During the COVID-19 pandemic lockdowns I have (virtually) encountered many people who cannot focus on their task because of the disturbance of other task notifications. They are checking what the breaking news are or just waiting for the responses from other people. Even the movie called The Social Dilemma [spoiler alert] warned about the fact the companies want to lock the user in their application and the users are paying with their attention.

My app does the same pattern as the best mobile apps, it ties the user to use the app all day long. But instead of wasting their limited time with the help of great machine learning techniques, it helps them to disconnect from those disturbing notifications and make them focus on their important tasks. The app provides the feeling they are doing something on their phone by growing a tree inside the app while they are focusing. The app gives an optional notification when the app launches to guide them to turn on the *Do Not Disturb* mode, so they do not forget it and they are not getting disrupted in their tasks.

The user could specify how much time they want to focus and what they are doing by tagging the current session. They can customize the tags or add new ones for their own needs. The work of the users is stored in the cloud. The users could visualize what they have done to help them manage their time and see where their time budgets are spent.

Figure 1: Overview of the application on four iPhone devices (12 Pro Max, 12 Pro, SE 2nd gen., 12 mini).

2.2. Features

Here is the list of the features:

- Schedule the tasks on your own pace.
- Earn rewards by finishing the tasks in time.
- Mark the tasks with tags to represent what you have done.
- Customize the tags for your own needs.
- See where your time budgets are spent.
- Learn your focus habits by measuring your progress.
- Never forget to enable the Do Not Disturb mode to control your time.
- Store your progress safely in the cloud.
- Share your progress on multiple devices.
- Do something on your phone without touching it by planting a virtual tree.
- Customize the app to make it your own.
- Stops you to scrolling down on your news feeds.

3. User documentation

The main purpose of the application to help you focus and manage your time better. Let us see how.

3.1. Installation

3.1.1. Install from source

You need to use the macOS operating system. You must download at least Xcode 12.5 from the App Store [1] (released on 26 April 2021) which will build and run for the iOS 14.5 target platform using the language Swift 5.4.

You need to obtain the source code of this thesis, and then open the main *Tree.xcodeproj* file which already specifies the correct settings. You need to pick a simulator or a real iOS device running with at least iOS 14.5 as the target device. Click on the build and run button on the top-left. After that, the application should run.

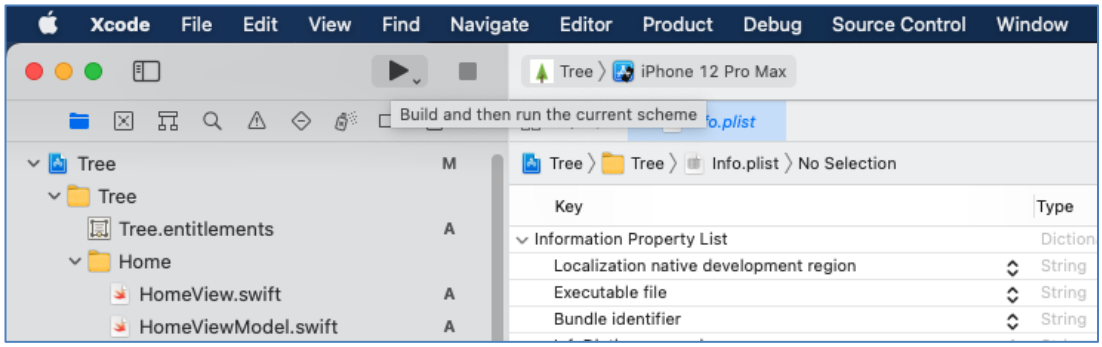


Figure 2: The build and run button in Xcode.

Some features are only available if you join in the Apple Developer Program [2] which is a paid membership.

3.1.2. Install from the App Store

The real polished application will be available for download directly from the App Store. It will support iOS 14.5, iPadOS 14.5, macOS Big Sur 11.3 and later versions. You just need to search for "*Tree time management*". Please refer to the official download guides to install the application [3].

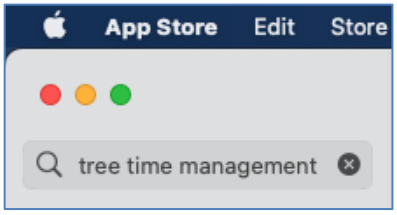


Figure 3: Searching for the application in the App Store.

3.2. Home screen

The main screen is the home screen, where you could start a new focus session with the *Play* button. You need to specify how much time do you want to focus by the slider on the bottom of the screen. During the session you can pause or stop the current session. The application supports two modes for your focusing: *Growing* mode and *Panicking* mode. The *Growing* mode is when you are do general tasks, and the timer will go forwards as time passing by. You could also pick the *Panicking* mode, which tries to simulate the situation when you are limited in time, like you are on an exam, and so that the timer will go backwards as you run out of time. For each session you could select an appropriate tag to tag your session so that later you can measure what you have achieved.

By using the application, you could earn experience points. The more experience points you acquire, the better you should feel about yourself. It represents your overall commitment to put down your phone, and just do whatever you wanted to do.

After each session there is 5 minutes to register that you have finished with your task and claim your rewards. If you do not confirm that you have finished with your task, the application assumes that you did not accomplish the task within the time you wanted to finish and left the task at some point. If that is the case, you earn the quartet of the experience points you would earn by finishing the task. You can pause or stop the session any time, and by stopping the task you earn all the experience points based on how much time you have focused so far. If you leave the app, the timer does not move forward. If you exit the app, you lose the progress of the current session all together.

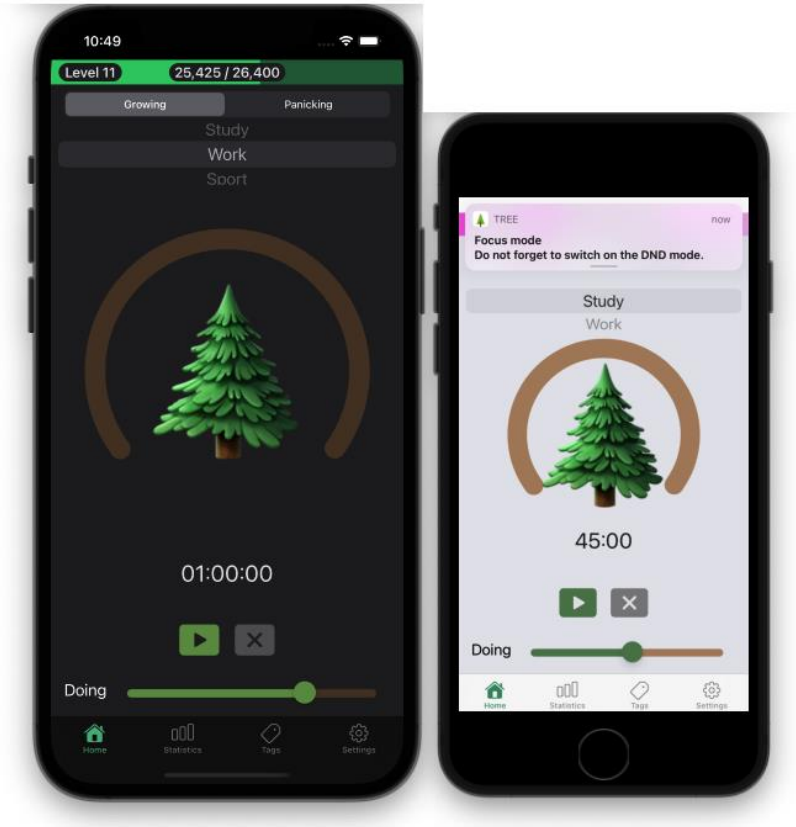


Figure 4: The home screen.

3.2.1. Notification

When the application launches you can get a reminder to enable the *Do Not Disturb* mode meanwhile you are focusing on your task. If you would like to enable this notification, please refer to the user documentation of the *Settings screen* [3.5].

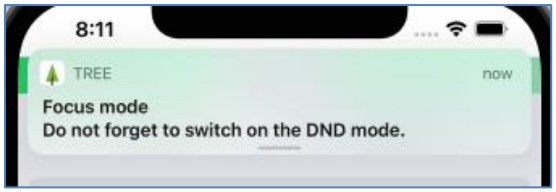


Figure 5: The DND mode reminder when the app launches.

3.2.2. Level progress

On the top of the screen, you can see your overall progress using the application. From the leading side of the screen to the trailing side of the screen you can see your current level, current experience points and the experience points needed for the next level.

In the background of this bar, you can see the current experience points and the next-level experience points ratio visualized. You could change the color of the bar in the settings menu to make the home screen your own.

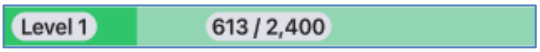


Figure 6: The level progress, where 613 experience points (xp) earned, 2400 xp required to level up.

3.2.3. Mode picker

On the top of the screen, you could switch between *Growing* and *Panicking* modes:

- *Growing mode*: The timer goes forwards to simulate that you are growing your skills. This is the default mode.
- *Panicking mode*: The timer goes backwards to simulate that you are running out of time and gives you the panicking feeling, like if you would attend a serious exam.

Figure 7: The mode picker, where the *Growing mode* is selected.

3.2.4. Tag picker

On the top of the screen, you could select the appropriate tag for the current task that you are trying to accomplish. The default task is the first task from your *Task list*. You can select the tag by swiping up and/or down on the picker, and then the selected tag will be highlighted in the middle of the picker. You can specify your own tags for your own needs in the *Tags screen* [3.4].

Figure 8: The tag picker, where the *Work tag* is selected.

3.2.5. Task progress

In the middle of the screen, you can see the progress of your task and the tree. During an active session the progress bar slowly progresses forwards or backwards based on the mode you have picked, and the tree grows. At the end of each task a new progress bar shows how much time left to claim your rewards.

Figure 9: The task progress, where 2/3 of the task is done.

3.2.6. Timers

On the bottom of the screen, you can see how much focus time is specified by the *Doing slider*. During an active session the timer count forwards of backwards based on the mode you have picked. When you have focused for the specified time amount a new timer starts just below the main timer and count 5 minutes backwards meanwhile you can register that you have finished with your task.

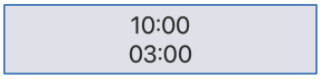


Figure 10: The timers, where a 10-minute task is done, and 3 minutes left to confirm the task.

3.2.7. Action buttons

On the bottom of the screen, you can see two buttons to start, pause, stop, and confirm the session. After tapping on the *Stop button* a pop-up will ask you whether you are truly want to give up your current task. The following list demonstrates each of the possible actions:

Before the session starts, you could begin it by tapping on the *Play button*.

During the active session, you could pause or stop the session.

During the paused session, you could continue or stop the session.

After the section finishes, you could confirm it by tapping on the *Done button*.

3.2.8. Doing slider

On the bottom of the screen, you could specify the committed duration for your task. By moving the slider to the right, the duration increases. The current selected duration is visible on the timer, just above the slider. The limit is an 80-minute-long task.

You can tap anywhere in the slider which will bring select the appropriate timer state. This is an additional feature for your user experience, which is not present in the default iOS, iPadOS and macOS sliders.

Figure 11: The doing slider, where a 20-minute task is set.

3.3. Statistics screen

The statistics screen gives you the overview about what you have done so far using the application. Here you can learn how you spend your time in terms of numbers. You can traverse backwards and forwards in time to measure your performance.

The application supports four types of measurement: daily, weekly, monthly, and yearly. You can traverse your task history by these date intervals to see the differences in your progress, whether you have spent your time better.

You can find two charts on this screen:

- Focused time chart: summarizes your finished task progression.
- Tag distribution chart: compares your finished tasks.

Figure 12: The statistics screen.

3.3.1. Date interval picker

On the top of the screen, you could pick from the following date intervals: *Day*, *Week*, *Month* or *Year*. The default selection is *Week*. By selecting the interesting one, you will see your progress for the entire interval. For example, if you pick the interval of *Week*, you will see your progress for the current week.

Figure 13: The date interval picker, where the Week is selected.

3.3.2. Navigation buttons

On the top of the screen, you could move backwards or forwards in time to measure your progress. Using the leading button, you go backwards in time. Using the trailing button, you go forwards in time. There is a *Reset button* in the middle part which brings you back to the present. The middle text let you know where you are in your timeline.

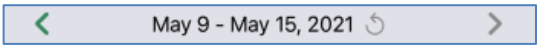


Figure 14: The navigation buttons, in order: backwards button, reset button, forwards button.

3.3.3. Focused time summary

On the top of the screen, it calculates the minutes spent in the application in the selected date interval. With that you can measure your progress on the same scale meanwhile you traverse in your past progression.

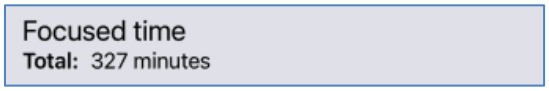


Figure 15: The focused time summary in minutes.

3.3.4. Focused time chart

It is in the top half part of the screen. It is a typical bar chart which let you know how much time you have focused so that you can measure your progress. You can tap on each bar to see the underlying numbers which they represent. You can also swipe on the bars (even outside of the bar chart) to compare your progress even faster.

Every bar is measured in minutes which is indicated at the bottom-leading corner by the *"0 M"* text, which stands for *"0 minutes"*. The largest measurement on the leading side is also the largest value in the bar chart. The leading side numbers are helping you to roughly see the values of the bars for easier measurement.

The bottom values (or tags) of the bar chart are different for all date intervals:

- Day: It represents the 0:00 to 23:59 interval for each hour.
- Week: It represents the Sunday to Saturday interval for each day.
- Month: It represents the first day to the last day of the month interval for each day.
- Year: It represents the January to December interval for each month.

If you select a bar it will be highlighted and its tag with its value in minutes will be visible on the top of the bar chart.

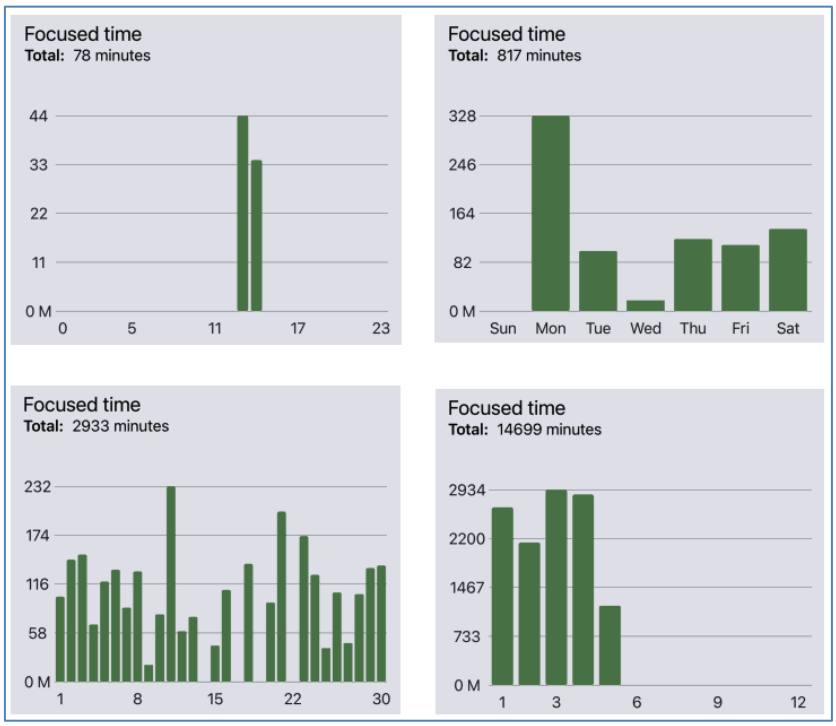


Figure 16: The four different date interval based focused time charts.

3.3.5. Tag distribution chart

It is in the bottom half of the screen. It is a typical donut chart which let you know on which scale you have spent your time for each kind of task. It is measured by the tags attached to the tasks and represented as percentiles compared to all the other tasks.

On the left side of the screen, you can see the donut chart to give you a visual representation of the tag distribution with the colors of the tags. You can tap on the donut slices. If you select a donut slice, its values will be visible on the top of the donut chart. By default, the most time-consuming task is visible.

On the right side of the screen, you can see the list of your tags with their scale as values with percentile. You can swipe the list up and down to see more of your tags. The list elements are ordered by the largest time-consuming task to the smallest by also ordering them according to your ordering in the tag list. The structure for each row is the following: the color of the tag, the scale, and the name of the tag.

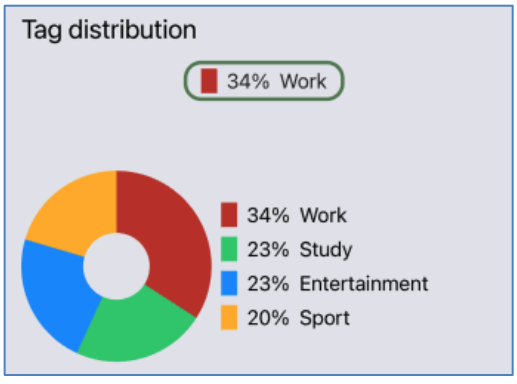


Figure 17: The tag distribution donut chart.

3.4. Tags screen

The tags screen gives you the opportunity to organize your tasks with tags. For regular tasks you are doing you can create new tags and on the home screen you can use them. Here you can also modify and delete existing tags.

When you tap on a tag, it will open a new screen to edit the selected tag. When you have finished with the customization of the selected tag, you just need to leave the tag editing screen to save your modifications.

This is the traditional iOS application list with the swipe-to-delete and editing mode support. In the editing mode you can rearrange the tags by importance.

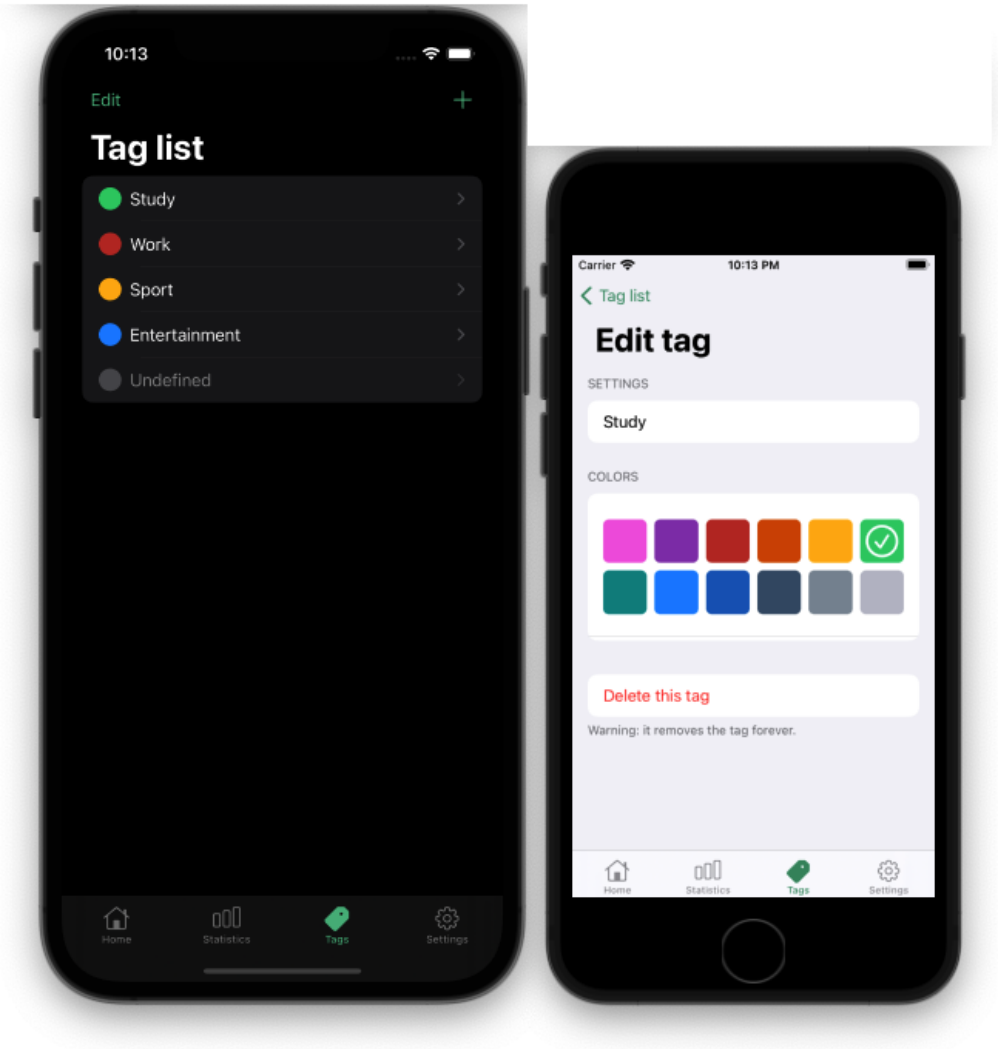


Figure 18: The tags screen with its edit tag screen.

3.4.1. Action buttons

On the top of the screen, there are two common buttons for list management. On the leading part of it you can find the *Edit button*, and on the trailing part of it you can find the *Add button* denoted by the plus sign (*"+"*).

The *Edit button* toggles the edit mode of the list. Using the edit mode, you can rearrange the order of the tags for your current needs. It is also a lot easier to delete the unneeded tags because each tag has a delete button.

The *Add button* creates a new tag with pink color and the name *"New tag"*. You cannot generate multiple new tags at once so that if you tap on the *Add button* multiple times by mistake you will still have only one new tag to customize.

Figure 19: The action buttons, in order: edit button, add button.

3.4.2. Tag list

Each tag row shows the color and the name of the tag. By tapping on any of the rows, you enter to the *Edit tag screen* [3.4.3] where you can customize the selected tag.

The last row is special, it defines the *Undefined tag* which you cannot modify or remove. It is useful if you do not want to tag your tasks.

Figure 20: The tag editing mode is disabled.

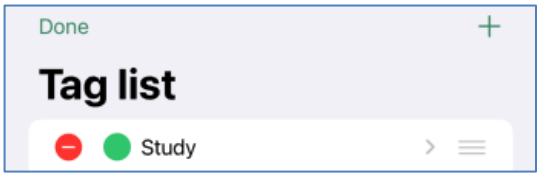


Figure 21: The tag editing mode is enabled.

3.4.3. Edit tag screen

Here you can edit the name and the color of the tag using 12 predefined colors. The selected color is indicated with a checkmark on it. If you do not need the tag, you can delete it using that edit menu as well.

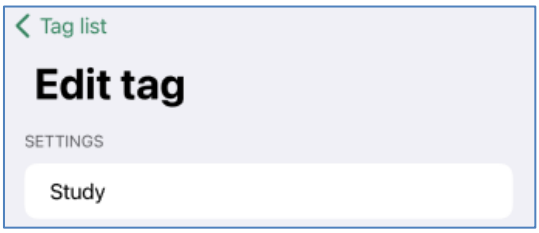


Figure 22: The edit tag screen changes the layout.

3.5. Settings screen

The settings menu gives you the opportunity to override the behavior of the application.

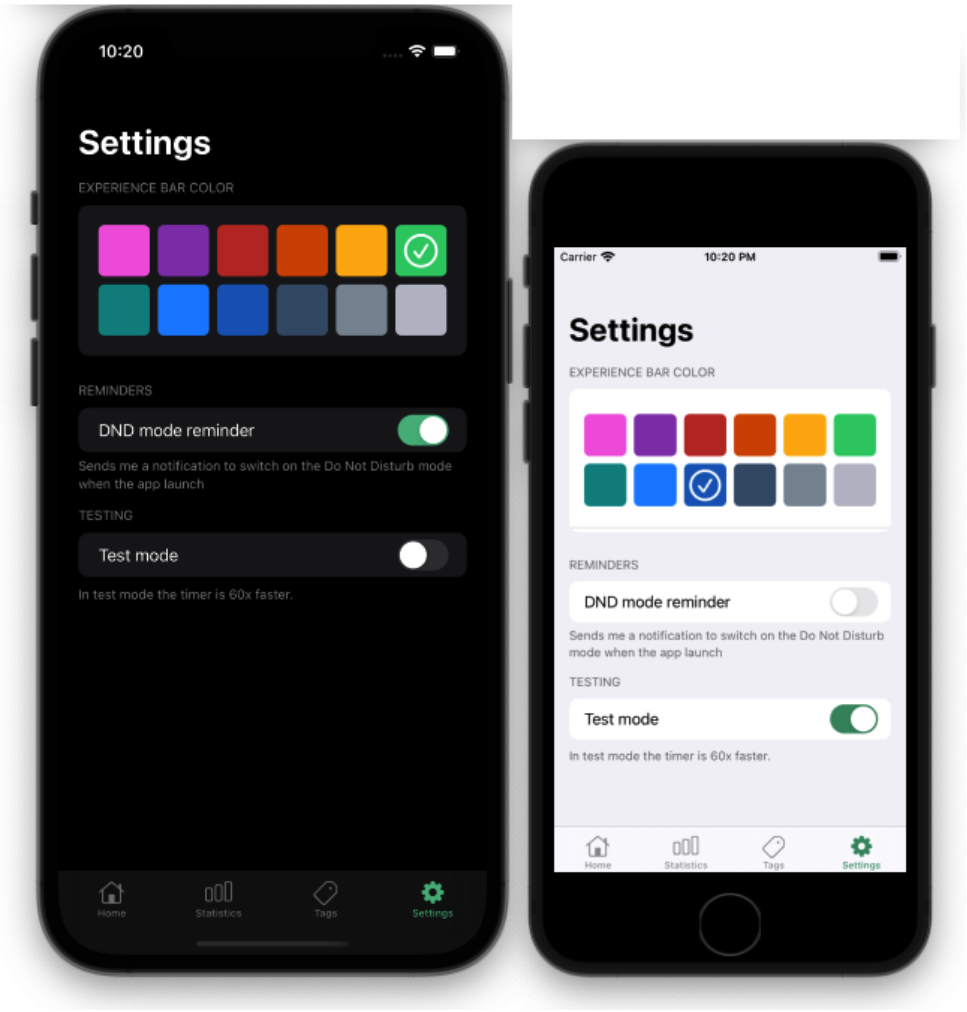


Figure 23: The settings screen.

3.5.1. Experience bar color

You can customize the color of the experience bar on the home screen. The selected color is indicated with a checkmark on it.

3.5.2. DND mode reminder

This toggles the reminders on the application launch to switch on the *Do Not Disturb* mode.

On the first enable it will ask for permissions whether you would like to permit the application to send you notifications.

3.5.3. Test mode

This toggles the test mode feature. The testing mode accelerates the timer by 60 times so that each minute of committed work is being done in a second. With that the minimum required time to commit is 5 seconds and you have 5 seconds to confirm the task. This feature is for demonstration purposes only.

4. Developer documentation

I have used the latest technologies and standards what Apple provides and working out of the box using Swift 5.4 and SwiftUI 2.0 in Xcode 12.5 [1] (released on 26 April 2021). I have not used any third-party dependency.

I have gathered all the available knowledge and best practices from multiple resources then I have combined them together to represent the state-of-the-art iOS development. After that I have injected my own game development background (Unity, Unreal Engine 4) to make the application even more *"pro"*.

Please note that, even the latest WWDC 2019 talks are outdated, because of the extremely fast development of Swift and SwiftUI. This documentation could be completely outdated by the time you read it, but if that would be the case, it gives you the overview of the standards just before the WWDC 2021 [4] announcements.

4.1. Analysis

In this section I present the development on the level of ideas, thoughts. It gives you the bedrock of the development.

4.1.1. Design patterns

Up until today (before the WWDC 2021 [4]) there is no Apple recommended design pattern when it comes to develop iOS applications using SwiftUI.

4.1.2. MVC design pattern

The Model-View-Controller design pattern [5] was the de facto standard for iOS development using Objective-C and people still use it. These MVC-based classes are huge because the model of the view is not entirely separated from the view. The application should try to avoid to directly use MVC, because the SwiftUI views should be as lightweight as possible. The SwiftUI views should require a very tiny business logic. Indirectly the application needs to use MVC ideas to separate the business logic from the view.

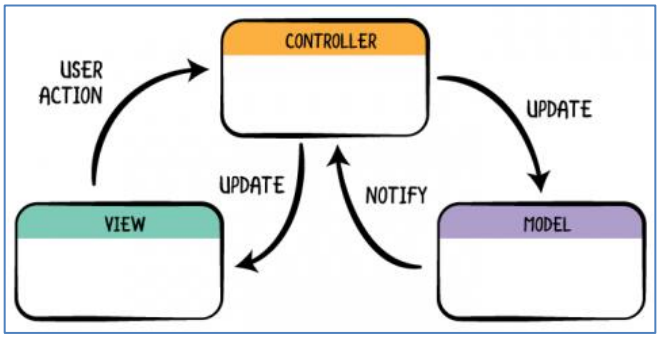


Figure 24: The MVC design pattern (source: raywenderlich [39]).

4.1.3. MVVM design pattern

The Model-View-ViewModel design pattern [6] is the evolved version of the MVC which could be applied for SwiftUI views. It is the trendy way to separate the business logic and somewhat overused for tiny views which would be simpler and more maintainable without MVVM. The application should use MVVM for large views. There is no direct support for the MVVM design pattern by SwiftUI, but it is sometimes worth the extra effort to separate the already huge views from their model, using the help of the separated view-model.

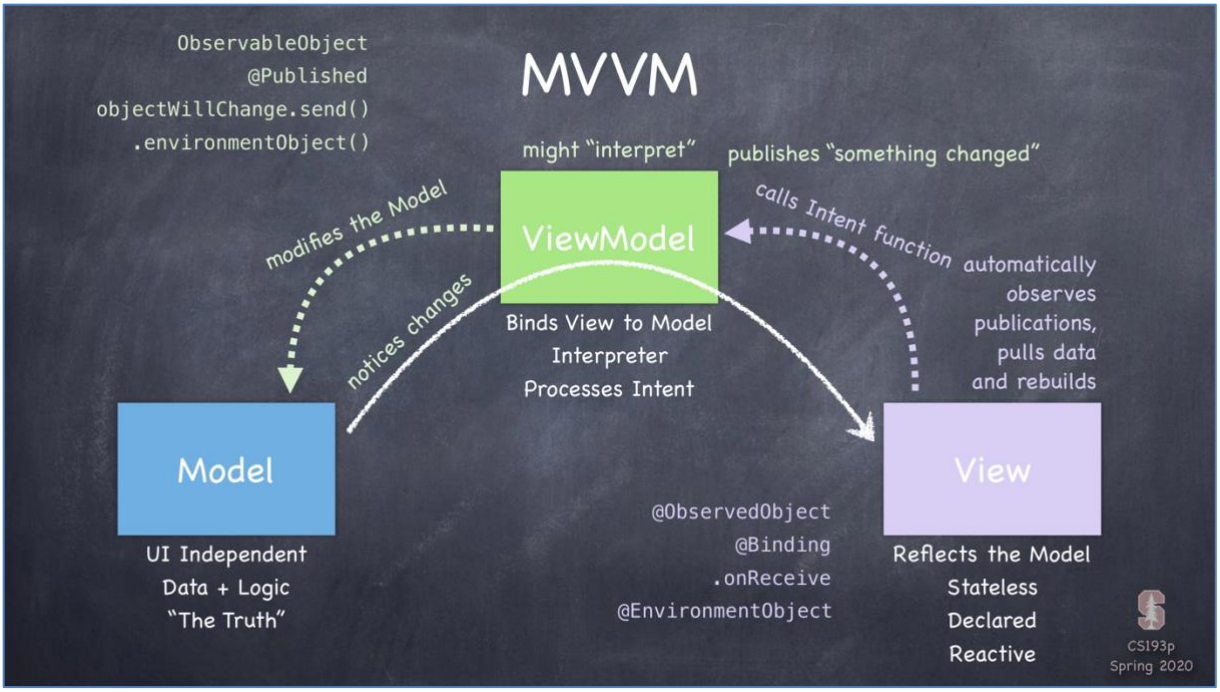


Figure 25: The MVVM design pattern (source: Stanford University CS193p 2020: Lecture 2 [7]).

4.1.4. Observer design pattern

The Observer design pattern [8] is heavily used by the SwiftUI views by the ObservableObject protocol [9] and the @Published property wrapper [10]. Using the MVVM design pattern the usual SwiftUI view ends up with only one observed object, the view-model, which is working most of the time. The problem with that is that, one observed object cannot subscribe to another observed object by default. The application needs to enhance the MVVM design pattern with another layer of the Observer design pattern [8] on top of the MVVM design pattern [6] to make the MVVM even more useful.

Figure 26: The Observer design pattern (source: raywenderlich [11]).

4.1.5. Handlers: Single-responsibility principle

In game development, the developers create separate handlers which are handling the experience points, handling the health, handling the energy, handling the gold in the game. The game developers just drag-and-drop the handlers in the UI of the game development framework to use the handlers in other handlers which are referring to one single instance of the handler like using the Singleton design pattern [12].

The handlers are useful for creating the Single-responsibility principle [13] in the extremely complicated games. This application to some degree implements a tiny game, so that it should use handlers for better maintainability and scalability.

In this heavily concurrent and stateful view setting I try to avoid the Singleton design pattern [12] as people believe it is the root of all evil and goes against the object-oriented software development standards [14].

The application could use the @EnvironmentObject property wrapper [15] to state out a global scope of the handler, but this property wrapper is an ObservableObject, and the problem is that, ObservableObjects could not listen to ObservableObject by the default design.

4.1.6. MVHVM design pattern

This is the theoretical name to use the handlers and the MVVM design pattern [6] together. It can be done using the Combine framework [16] by combining the handler and the view-model classes.

4.1.7. Views

It is that easy to create the views with the SwiftUI framework it takes more time to come up with wireframes or mockups than to create the real production views, therefore I do not provide mockups.

During the application development the preview immediately updates on code changes. We can also try the application in action using the live preview mode.

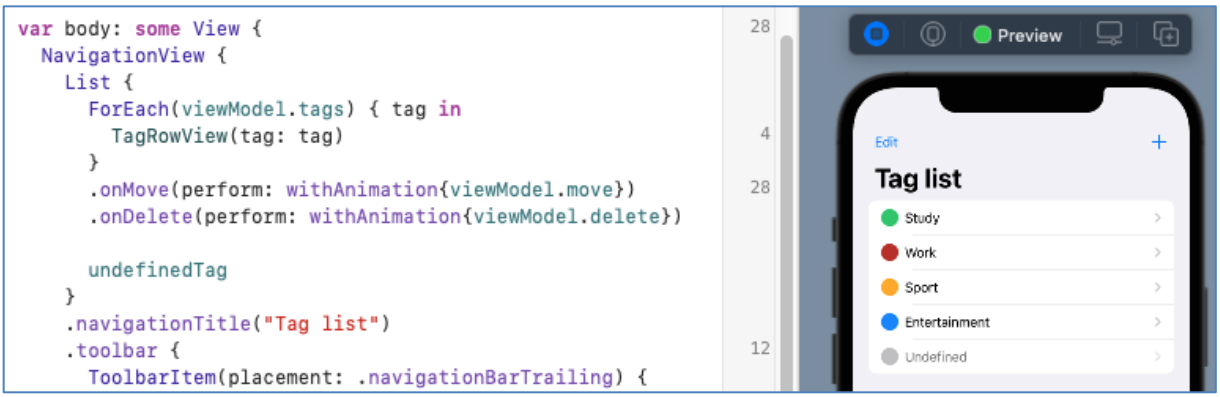


Figure 27: The way of working with SwiftUI views using the live preview feature.

4.2. Design

In this section I present the core facilities which is the bedrock of the application.

4.2.1. Data handling

The users should be able to store their progress, the `Tasks`, and the associated `Tags` of the tasks in a cloud database.

The following table defines the data model of the `Task`:

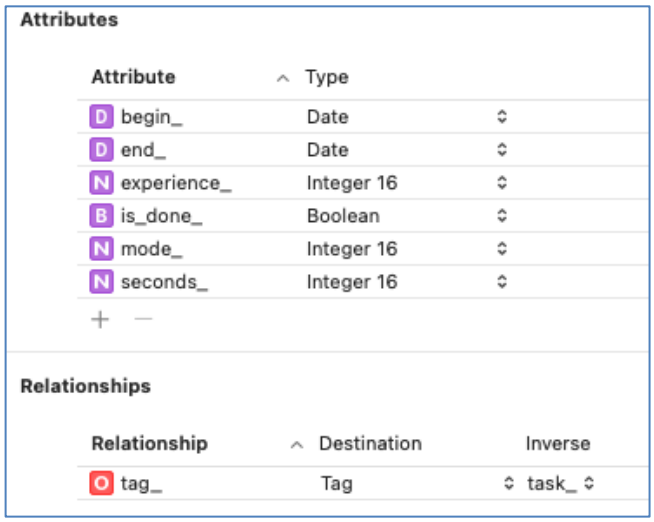


Figure 28: `Task` entity in the *Main.xcdatamodeld* file.

The following table defines the data model of the `Tag`:

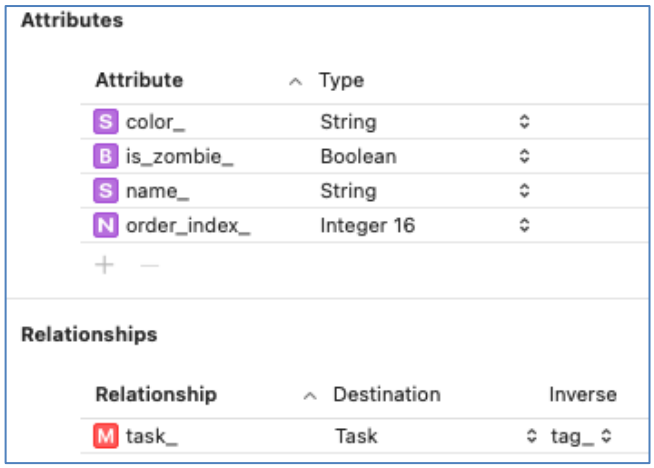


Figure 29: `Tag` entity in the *Main.xcdatamodeld* file.

4.2.2. Time handling

The application needs to track how much time the user wants to allocate, and how much progress they are making. The user should be able to start, pause, stop, and confirm their progression. The application needs to use a `Timer` [17] which does not run in the background and does not listen to the device time changes. With that the user cannot cheat the time: they must run the application without surfing the web, and if they change the time settings, nothing happens in the state of the timer.

The applications must give the user the time to register that they have accomplished their task and does not leave their phone too far away. If they have finished with their time commitment, the timer should represent the overflow state and delegate the state of the timer to the overflow time stream.

The following table defines the states of the time handler:

None	When the timer is inactive: the user has not started the timer.
Active(Normal)	When the timer is active and normal: the user is working on their task.
Active(Overflow)	When the timer is active and overflowing: the user needs to confirm the task.
Pause	When the timer is paused.
Expired	When the specified time overflow limit invalidates the timer.

4.2.3. Date handling

The application needs to traverse the dates for the settings view on four different scales, using daily, weekly, monthly, and yearly intervals.

When the user selects the interesting interval for their needs the application should gather all the tasks belonging to the selected interval. To support that the date handler needs to use the `Calendar.enumerateDates()` API [18] and call a callback during the iteration. This API is difficult to use, because it does not start from the specific date, but rather after that specific date. With that the data handler must assemble the dates by specific granularities of date components using the selected interval.

For example: let assume that the date handler needs to traverse a given year which means it needs to hit 12 months. The start component is December of the previous year, which is specified as a month-level granularity offset by subtracting it from the original year. The range to traverse is a year-level granularity component.

The entire interval is specified as:

$$
\begin{aligned}
startDate &= selectedDate(startComponents) \\
startDate &= startDate - offsetComponent \\
endDate &= startDate + rangeComponent \\
interval &= (startDate, endDate)
\end{aligned}
$$

Equation 1: Calculation of the date interval to make its range traversable by the *Calendar API* [19].

The same logic applies to all the interval kinds.

4.2.4. Task handling

The application needs to track the iCloud database to fetch, upload and sync the tasks of the user. There is a designated API for that called `@FetchRequest` [20], but it is not working outside of views. Instead of this API the `NSFetchedResultsController` [21] does the same job and working in both view-models, handlers, and views as well.

4.2.5. Combine the handlers and view-models

To enable the facilities like the handlers and view-models to communicate with each other we need to use the Combine framework [16] which combines event-processing operations.

The Foundation framework [22] already implements the core facilities using the Combine framework. When we implement SwiftUI code, we already write Combine code under the hood. We just need to create the connections when we create the object using the constructors.

4.2.6. Test handling

The application should run differently if it is being run as part of the user interface (UI) tests. It should clean up the mess by the previous UI test for making each test self-containing.

The UI tests are slow and here are especially slow because this application needs time to progress with the tasks. In test mode we need to switch off the animations to make the UI respond faster and so that the tests are passing faster.

4.2.7. View handling

The views need to agree how much space the sub-views should take. Sometimes the `Text` views [23] cannot show the entire content because we apply the view modifiers in a wrong order, or not at all. The best practice is to force the view to require as much space as needed by the `fixedSize()` view modifier [24].

If there are multiple views arranged in some relationship, for example below each other, and our task is to show the full data of each of the rows, then we are doomed by the default view hierarchy. In that scenario one of the rows will require some space, and the entire column will take up that much space, truncating the data of all the other rows. We can fix that in two ways:

- We could create an invisible static row which takes the value of the widest row, and this forces the view hierarchy to show all the data. This is the common approach because it is easy to use and easy to reason about.
- We could interoperate with the view hierarchy and explicitly state out the size needs of the view with the preference view state modifier [25].

Each of these methods has the pros and cons, but the result is the same.

4.2.8. Notification handling

The application needs to request permission from the user to present notifications. For the better user experience, it should not block the user to use the application. The permission request should happen when the user wants to enable the *Do Not Disturb* mode notifications using the settings screen of the application.

If the application is active and it emits a notification, then the notification fires in the background and will not present to the user by default. The application should override the default behavior and present the notification meanwhile the application is running, when the application launched.

4.3. Implementation

In this section I present the programming challenges to implement the application.

4.3.1. Data handling

For each Core Data component, the application should relax the raw optional and differently typed iCloud database values. These helper routines are defined in the *Data* folder.

Here are two examples of such helper computed properties:

```
extension Tag {
  var name: String {
    get { name_ ?? "Undefined" }
    set { name_ = newValue }
  }

  var orderIndex: Int {
    get { Int(order_index_) }
    set { order_index_ = Int16(newValue) }
  }
}
```

Figure 30: Code snippet to work with raw iCloud values (source: *Tag+CoreDataHelper.swift*).

If the user deletes tags the database still must store the tags because the statistics view should be able to represent all the user activity. For that instead of removing the tags, the Core Data Tag helper function `delete()` will mark the Tag as a *zombie*. Such *zombie* Tags are considered as deleted ones, but the statistics view could list them appropriately.

It is possible to override the default Core Data `delete()` method, but for testing purposes the application should be able to truly remove the tags.

In the `@FetchRequest` property wrapper [20] we have to define the `sortDescriptors` argument which takes multiple `NSSortDescriptors` [26] as an argument. To sort the Core Data objects in the request we could not use the non-optional helper routines, we must explicitly use the generated optional `KeyPaths` [27]. In this property wrapper we could define the `NSPredicate` [28] of the request, which takes a custom database language format [29] for filtering objects.

Here is an example:

```
  @FetchRequest(entity: Task.entity(), sortDescriptors: [
    NSSortDescriptor(keyPath: \Task.begin_, ascending: true)
  ], predicate: NSPredicate(format: "is_zombie_ = FALSE"))
  var tasks: FetchedResults<Task>
```

Figure 31: Code snippet to fetch and sync with the iCloud database.

4.3.2. Time handling

This is defined in the *Handler* folder, in the *TimeHandler.swift* source file. The most important part is the state of the timer so that listeners to that class could act upon the state changes.

The state is defined using two `enums`, where the main `TimerState enum` contains the *associated value* of the `TimerActiveState enum` [30] where appropriate:

```
enum TimerActiveState { case Normal, Overflow }
enum TimerState: Equatable {
  case None, Active(TimerActiveState), Pause, Expired
}
```

Figure 32: Code snippet to declare *associated values* in `enums` [30] (source: *TimeHandler.swift*).

In the core logic of the time handler there are `DEBUG` *compilation conditions* [31] so that the release build will not contain the testing code. This is a very ugly but very effective way to help the compiler to optimize the code for the release. Here is an example about this feature and about the usage of the associated values of `enums`:

```
#if DEBUG
import SwiftUI
#endif

class TimeHandler: ObservableObject {
  #if DEBUG
  @AppStorage("isTestMode") private var isTestMode = false
  #endif

  private var timer = Timer()

  func run() {
    tryStateUpdate(.Active(.Normal))

    timer =
      Timer.scheduledTimer(withTimeInterval: 1, repeats: true) { [self] _ in
        switch timerState {
          case .Active(.Normal):
            seconds += 1
            #if DEBUG
            if isTestMode {
              seconds += 59
            }
            #endif
            if seconds >= endSeconds {
              tryStateUpdate(.Active(.Overflow))
            }

          // ...
        }
      }
  }
}
```

Figure 33: Code snippet to use *compilation conditions* and *associated values* (source: *TimeHandler.swift*).

4.3.3. Date handling

This is defined in the *Handler* folder, in the *DateHandler.swift* source file. The core part of this class is the `selectedDateInterval` computed property which makes the date traversal possible. It uses the `Calendar` and its extremely powerful API [19] which is instantiated inline using the following closure syntax (which is not that well known):

```
class DateHandler: ObservableObject {
  let calendar: Calendar = {
    var calendar = Calendar(identifier: .gregorian)
    calendar.locale = NSLocale(localeIdentifier: "en_US") as Locale
    return calendar
  }()

  private func selectedDateInterval(
    startComponents: Set<Calendar.Component>,
    offsetComponent: Calendar.Component,
    rangeComponent: Calendar.Component) -> DateInterval {
    var startDate = calendar.date(
      from: calendar.dateComponents(startComponents, from: selectedDate))!
    startDate = calendar.date(
      byAdding: offsetComponent, value: -1, to: startDate)!
    let endDate = calendar.date(
      byAdding: rangeComponent, value: 1, to: startDate)!

    return DateInterval(start: startDate, end: endDate)
  }

  var selectedDateInterval: DateInterval {
    switch selectedInterval {
      case .Year:
        return selectedDateInterval(
          startComponents: [.year],
          offsetComponent: .month,
          rangeComponent: .year)
      // ...
    }
  }
}
```

Figure 34: Code snippet to use *inline constructors* and the `Calendar` [19] (source: *DataHandler.swift*).

4.3.4. Task handling

This is defined in the *Handler* folder, in the *TaskHandler.swift* source file. The core component is the automatically fetching tasks which is following the upstream iCloud database changes and keeps the tasks in sync. By changing the convenient `@FetchRequest` [20] iCloud database sync code (being used in views) to that big chunk of code we can use the same sync strategy outside of the views:

```
class TaskHandler: NSObject, ObservableObject,
                   NSFetchedResultsControllerDelegate {
  let dataController: DataController

  private let tasksController: NSFetchedResultsController<Task>
  @Published var tasks = [Task]()

  init(dataController: DataController) {
    self.dataController = dataController

    let request: NSFetchRequest<Task> = Task.fetchRequest()
    request.sortDescriptors =
      [NSSortDescriptor(keyPath: \Task.begin_, ascending: true)]
    request.predicate = NSPredicate(format: "is_done_ = TRUE")

    tasksController = NSFetchedResultsController(fetchRequest: request,
      managedObjectContext: dataController.container.viewContext,
      sectionNameKeyPath: nil, cacheName: nil)

    super.init()
    tasksController.delegate = self

    try! tasksController.performFetch()
    tasks = tasksController.fetchedObjects ?? []
  }

  func controllerDidChangeContent(
    _ controller: NSFetchedResultsController<NSFetchRequestResult>) {
    if let newTasks = controller.fetchedObjects as? [Task] {
      tasks = newTasks
    }
  }
}
```

Figure 35: Code snippet to create iCloud database sync outside of the views (source: *TaskHandler.swift*).

4.3.5. Combine the handlers and view-models

The original source code snippet [32] to combine handlers and the view-models with the `Combine` framework [16] has a bug. The view-model layer updates the view, which should happen in the main thread. By extending the original source code, the home view-model could listen and publish the changes of the time handler:

```
extension HomeView {
  class ViewModel: NSObject, ObservableObject,
                   NSFetchedResultsControllerDelegate {
    let 🌳 = "🎄" /// So pro.

    init() {
      timeHandler.objectWillChange
        .receive(on: DispatchQueue.main)
        .sink{_ in self.objectWillChange.send()}
        .store(in: &cancellables)
    }
  }
}
```

Figure 36: Code snippet to combine multiple `ObservableObjects` [9] (source: *HomeViewModel.swift*).

4.3.6. Test handling

When we create the UI test cases, we need to tell the application to run in test mode. The test mode should be enabled for all the test cases. Therefore, I have created a class which will create the application instance with testing enabled, and all the other test cases should inherit from that test case and use this application instance:

```
class BaseUITest: XCTestCase {
  var app: XCUIApplication!

  override func setUpWithError() throws {
    app = XCUIApplication()
    app.launchArguments = ["enable-testing"]
    app.launch()
  }
}
```

Figure 37: Code snippet to use app launch arguments (source: *BaseUITest.swift*).

The application cleans up the changes from the previous test case and disables the animations when the `DataController` is being created and the store is loaded:

```
class DataController: ObservableObject {
  init() {
    // ...
    container.loadPersistentStores { _, _ in
      #if DEBUG
      if CommandLine.arguments.contains("enable-testing") {
        self.deleteAll()
        UIView.setAnimationsEnabled(false)
      }
      #endif
    }
  }
}
```

Figure 38: Code snippet to read app arguments and disable animations (source: *DataController.swift*).

4.3.7. View handling

I wanted to create a table view for the list of the donut chart values in the statistics screen. Because there is no table view exist, I ended up creating a list where I have explicitly specified the needed width for representing a column of values. The second column is aligned and so that the texts:

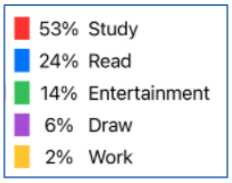


Figure 39: The tag distribution is aligned into three columns.

The original source code [33] to make this possible has a bug, because we need to mark the text with the `fixedSize()` view modifier [24] in order to preserve its content:

```
struct CenteringColumnPreference: Equatable {
  let width: CGFloat
}

struct CenteringColumnPreferenceKey: PreferenceKey {
  static var defaultValue: [CenteringColumnPreference] = []

  static func reduce(value: inout [CenteringColumnPreference],
                     nextValue: () -> [CenteringColumnPreference]) {
    value.append(contentsOf: nextValue())
  }
}

struct CenteringView: View {
  var body: some View {
    GeometryReader { geometry in
      Rectangle()
        .fill(Color.clear)
        .preference(
          key: CenteringColumnPreferenceKey.self,
          value: [CenteringColumnPreference(
                    width: geometry.frame(in: CoordinateSpace.global).width)]
        )
    }
  }
}

struct PieChart: View {
  @State private var percentileWidth: CGFloat? = nil

    Text(NumberFormatter.percentile(slice.pie.value / sum))
      .fixedSize()
      .frame(width: percentileWidth)
      .background(CenteringView())
      .onPreferenceChange(CenteringColumnPreferenceKey.self) { preferences in
        for preference in preferences {
          let oldWidth = percentileWidth ?? CGFloat.zero
          if preference.width > oldWidth {
            percentileWidth = preference.width
          }
        }
      }
}
```

Figure 40: Code snippet to specify the width of a column of texts (source: *PieChart.swift*).

4.3.8. Notification handling

The notifications are disabled by default. This setting is stored in the user settings using the `@AppStorage` property wrapper [34]. First, we need to request the permission to show notifications. If any error occurs during the request, we need to point the user to the device settings screen so they can enable the notifications as a device level setting for the application. Here is the huge boilerplate to do so:

```
struct SettingsView: View {
  @AppStorage("isReminderActive") var isReminderActive = false

  var body: some View {
          Toggle("DND mode reminder",
                 isOn: $isReminderActive.animation().onChange(notify))
  }

  func notify() {
    if isReminderActive {
      tryRequestNotifications { isSuccess in
        if !isSuccess {
          isReminderActive = false
        }
      }
    } else {
      UNUserNotificationCenter.current().removeAllPendingNotificationRequests()
    }
  }

  func tryRequestNotifications(completion: @escaping (Bool) -> Void) {
    UNUserNotificationCenter.current().getNotificationSettings { settings in
      switch settings.authorizationStatus {
        case .notDetermined:
          requestNotifications { isSuccess in
            DispatchQueue.main.async{completion(isSuccess)}
          }
        case .authorized:
          DispatchQueue.main.async{completion(true)}
        default:
          DispatchQueue.main.async{completion(false)}
      }
    }
  }

  private func requestNotifications(completion: @escaping (Bool) -> Void) {
    let center = UNUserNotificationCenter.current()

    center.requestAuthorization(options: [.alert, .sound]) { isGranted, _ in
      completion(isGranted)
    }
  }

  func showAppSettings() {
    guard let settingsURL = URL(string: UIApplication.openSettingsURLString)
    else { return }

    if UIApplication.shared.canOpenURL(settingsURL) {
      UIApplication.shared.open(settingsURL)
    }
  }
}
```

Figure 41: Code snippet to request permission to emit notifications (source: *SettingsView.swift*).

When the application launches, and the notifications are wanted we need to create it and present to the user:

```
@main
struct TreeApp: App {
  @AppStorage("isReminderActive") var isReminderActive = false

  var body: some Scene {
    WindowGroup {
      ContentView()
        .onAppear(perform: tryNotify)
    }
  }

  private func tryNotify() {
    if !isReminderActive {
      return
    }

    UNUserNotificationCenter.current().getNotificationSettings { settings in
      if settings.authorizationStatus == .authorized {
        notify()
      }
    }
  }

  private func notify() {
    let content = UNMutableNotificationContent()
    content.title = "Focus mode"
    content.subtitle = "Do not forget to switch on the DND mode."

    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 1,
                                                    repeats: false)

    let request = UNNotificationRequest(identifier: "id",
                                        content: content,
                                        trigger: trigger)

    UNUserNotificationCenter.current().add(request)
  }
}
```

Figure 42: Code snippet to emit notifications when the application launches (source: *TreeApp.swift*).

4.4. Testing

In this section I present the testing plan and the testing facilities.

4.4.1. Testing plan

Apple supports unit tests and UI tests [35]. They are complementing each other, but I think the UI tests are covering lot more than multiple unit tests could do, for example to measure the time it truly uses the Timer [17]. Therefore, I have created more UI tests than to create unit tests.

4.4.2. Unit tests

Here is the list of the unit tests which can be found in the *TreeTests* folder:

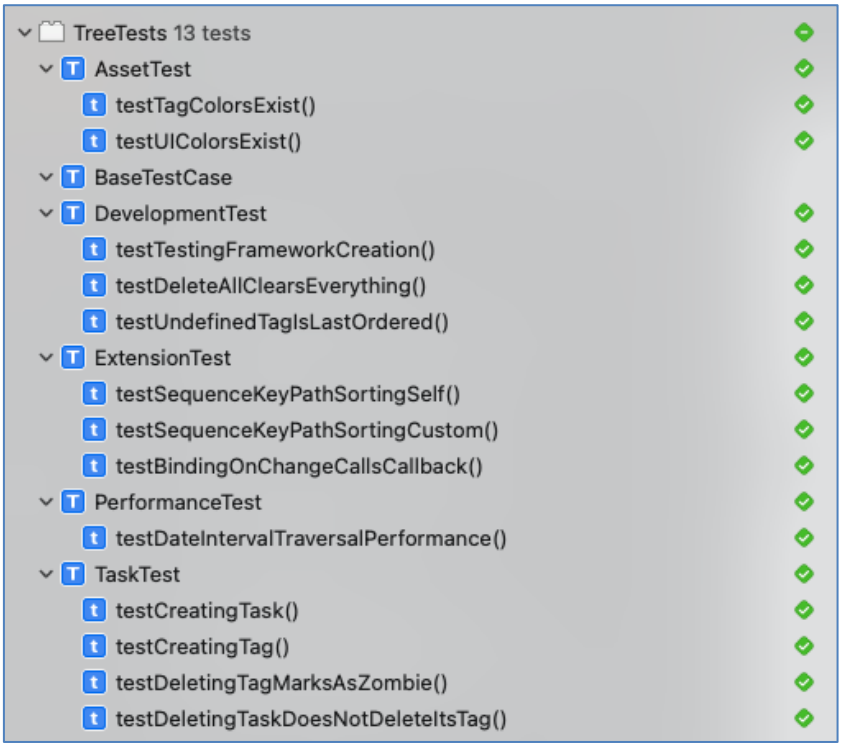


Figure 43: Overview of the passed unit tests.

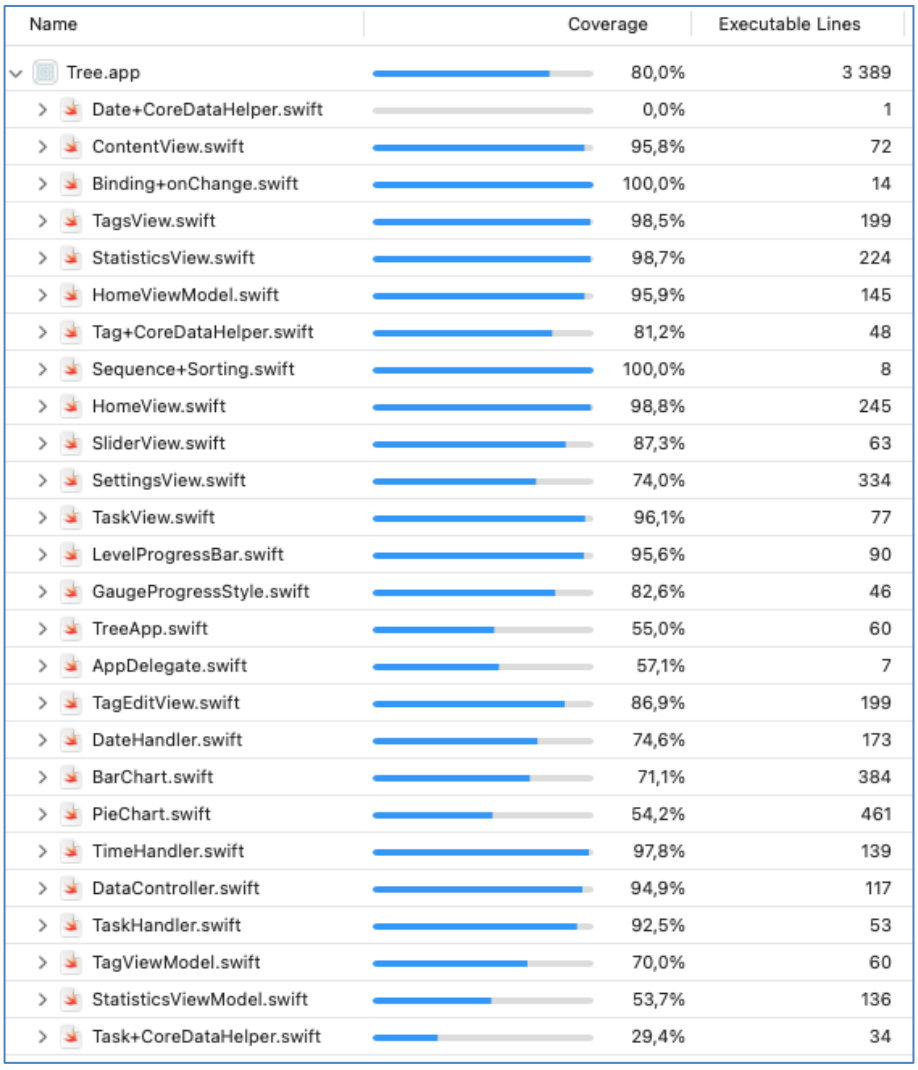
4.4.3. UI tests

Here is the list of the UI tests which can be found in the *TreeUITests* folder:

Figure 44: Overview of the passed UI tests.

4.4.4. Test coverage

I have tested the hot paths and I have not measured the failure-case conditions yet, therefore I have achieved a solid 80% test coverage in summary. I leave the chaos engineering exercises [36] for the reader to measure the failure paths as well.



Name	Coverage	Executable Lines
Tree.app	80,0%	3 389
Date+CoreDataHelper.swift	0,0%	1
ContentView.swift	95,8%	72
Binding+onChange.swift	100,0%	14
TagsView.swift	98,5%	199
StatisticsView.swift	98,7%	224
HomeViewModel.swift	95,9%	145
Tag+CoreDataHelper.swift	81,2%	48
Sequence+Sorting.swift	100,0%	8
HomeView.swift	98,8%	245
SliderView.swift	87,3%	63
SettingsView.swift	74,0%	334
TaskView.swift	96,1%	77
LevelProgressBar.swift	95,6%	90
GaugeProgressStyle.swift	82,6%	46
TreeApp.swift	55,0%	60
AppDelegate.swift	57,1%	7
TagEditView.swift	86,9%	199
DateHandler.swift	74,6%	173
BarChart.swift	71,1%	384
PieChart.swift	54,2%	461
TimeHandler.swift	97,8%	139
DataController.swift	94,9%	117
TaskHandler.swift	92,5%	53
TagViewModel.swift	70,0%	60
StatisticsViewModel.swift	53,7%	136
Task+CoreDataHelper.swift	29,4%	34

Figure 45: Test coverage (source: *Report navigator*).

5. Future work

Here are some ideas to make the application better:

- Accessibility: Everyone should be able to use the application. Sadly, I could not allocate enough time for that development.
- Default tags: When the app launches for the very first time it should provide useful default tags (like Study, Work, Sport and Entertainment).
- Cross platform support: The application should run on iOS, iPadOS and macOS.
- Extended cross platform support: It would be great if the application could run on Android devices and if it could run in internet browsers as a web application.
- Localization: The application should represent the dates in a localized format and should provide settings to customize the date format.
- Gold: The user should gain some in-game currency with the experience points, like the MMORPG games.
- More trees: The user should be able to buy new kind of trees for gold so they can plant the tree they like.
- Dying trees: If the user does not complete the task, they could receive a dead tree to inspire them to complete the task.
- Multiple UI: The application should adapt for multiple user interface layouts for better user experience.

6. Summary

What I have specified as the necessary work for my thesis is done. The application supports both light and dark modes so that it is looking great.

The iOS application development with SwiftUI is convenient until some point. This application demonstrates the extremely powerful Foundation APIs [22] as well as the weaknesses of SwiftUI.
To use MVVM [6] with the SwiftUI views we need to pay a little bit too much. We have seen that how to mix the MVC and MVVM design patterns in an effective way.

I hope the source code and the ideas are helpful for anyone who wants to see an end-to-end iOS development project and they can create even better projects using this thesis.

Stay hungry. Stay foolish. [37]

7. References

[1] "Xcode on App Store" [Online]. Available: https://apps.apple.com/us/app/xcode/id497799835. [Accessed 13. 05. 2021].

[2] "Apple Developer Program" [Online]. Available: https://developer.apple.com/programs/. [Accessed 13. 05. 2021].

[3] "Apple Support: Download apps and games on your iPhone or iPad" [Online]. Available: https://support.apple.com/en-us/HT204266. [Accessed 13. 05. 2021].

[4] "Apple Worldwide Developers Conference (WWDC) 2021" [Online]. Available: https://developer.apple.com/wwdc21/. [Accessed 13. 05. 2021].

[5] "Model–view–controller" [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. [Accessed 13. 05. 2021].

[6] "Model–view–viewmodel" [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel. [Accessed 13. 05. 2021].

[7] "Stanford University - CS193p - Developing Apps for iOS online lectures" [Online]. Available: https://cs193p.sites.stanford.edu/. [Accessed 13. 05. 2021].

[8] "Observer pattern" [Online]. Available: https://en.wikipedia.org/wiki/Observer_pattern. [Accessed 13. 05. 2021].

[9] "ObservableObject" [Online]. Available: https://developer.apple.com/documentation/combine/observableobject. [Accessed 13. 05. 2021].

[10] "Published" [Online]. Available: https://developer.apple.com/documentation/combine/published. [Accessed 13. 05. 2021].

[11] "Design Patterns by Tutorials book: Observer Pattern" [Online]. Available: https://www.raywenderlich.com/books/design-patterns-by-tutorials/v3.0/chapters/8-observer-pattern. [Accessed 13. 05. 2021].

[12] "Singleton pattern" [Online]. Available: https://en.wikipedia.org/wiki/Singleton_pattern. [Accessed 13. 05. 2021].

[13] "Single-responsibility principle" [Online]. Available: https://en.wikipedia.org/wiki/Single-responsibility_principle. [Accessed 13. 05. 2021].

[14] "C++ Core Guidelines: Avoid singletons" [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-singleton. [Accessed 13. 05. 2021].

[15] "EnvironmentObject" [Online]. Available: https://developer.apple.com/documentation/swiftui/environmentobject. [Accessed 13. 05. 2021].

[16] "Combine" [Online]. Available: https://developer.apple.com/documentation/combine. [Accessed 13. 05. 2021].

[17] "Timer" [Online]. Available: https://developer.apple.com/documentation/foundation/timer. [Accessed 13. 05. 2021].

[18] "Calendar.enumerateDates()" [Online]. Available: https://developer.apple.com/documentation/foundation/calendar/2293661-enumeratedates. [Accessed 13. 05. 2021].

[19] "Calendar" [Online]. Available: https://developer.apple.com/documentation/foundation/calendar. [Accessed 13. 05. 2021].

[20] "FetchRequest" [Online]. Available: https://developer.apple.com/documentation/swiftui/fetchrequest. [Accessed 13. 05. 2021].

[21] "NSFetchedResultsController" [Online]. Available: https://developer.apple.com/documentation/coredata/nsfetchedresultscontroller. [Accessed 13. 05. 2021].

[22] "Foundation framework" [Online]. Available: https://developer.apple.com/documentation/foundation. [Accessed 13. 05. 2021].

[23] "Text" [Online]. Available: https://developer.apple.com/documentation/swiftui/text. [Accessed 13. 05. 2021].

[24] "View modifiers: fixedSize()" [Online]. Available: https://developer.apple.com/documentation/swiftui/text/fixedsize(). [Accessed 13. 05. 2021].

[25] "View state modifier: preference()" [Online]. Available: https://developer.apple.com/documentation/swiftui/view/preference(key:value:). [Accessed 13. 05. 2021].

[26] "NSSortDescriptor" [Online]. Available: https://developer.apple.com/documentation/foundation/nssortdescriptor. [Accessed 13. 05. 2021].

[27] "KeyPath" [Online]. Available: https://developer.apple.com/documentation/swift/keypath. [Accessed 13. 05. 2021].

[28] "NSPredicate" [Online]. Available: https://developer.apple.com/documentation/foundation/nspredicate. [Accessed 13. 05. 2021].

[29] "Realm: NSPredicate Cheatsheet" [Online]. Available: https://academy.realm.io/posts/nspredicate-cheatsheet/. [Accessed 13. 05. 2021].

[30] "Swift Enumerations - Associated Values" [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID148. [Accessed 13. 05. 2021].

[31] "Swift Statements - Compiler Control Statements / Conditional Compilation Block" [Online]. Available: https://docs.swift.org/swift-book/ReferenceManual/Statements.html#ID538. [Accessed 13. 05. 2021].

[32] "Stack Overflow: Passing an ObservableObject model through another ObObject?" [Online]. Available: https://stackoverflow.com/a/59693022. [Accessed 13. 05. 2021].

[33] "Stack Overflow: Align two SwiftUI text views in HStack with correct alignment" [Online]. Available: https://stackoverflow.com/a/57677582. [Accessed 13. 05. 2021].

[34] "AppStorage" [Online]. Available: https://developer.apple.com/documentation/swiftui/appstorage. [Accessed 13. 05. 2021].

[35] "WWDC 2019: Testing in Xcode" [Online]. Available: https://developer.apple.com/videos/play/wwdc2019/413/. [Accessed 13. 05. 2021].

[36] "Principles of chaos engineering" [Online]. Available: https://principlesofchaos.org/. [Accessed 13. 05. 2021].

[37] "2005 Stanford Commencement Address delivered by Steve Jobs" [Online]. Available: https://news.stanford.edu/2005/06/14/jobs-061505/. [Accessed 13. 05. 2021].

[38] "Diagram of interactions within the MVC pattern" [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#/media/File:MVC-Process.svg. [Accessed 13. 05. 2021].

[39] "Model-View-Controller (MVC) in iOS – A Modern Approach" [Online]. Available: https://www.raywenderlich.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach. [Accessed 13. 05. 2021].

8. Figures