Diary Project

CSCU9YH – Android Development

2718715

2021

Table of Contents

[2 Appendix of Code 2](#_Toc90067544)

[3 The Process 3](#_Toc90067545)

[Introduction 3](#_Toc90067546)

[1. Storing the Entries - Database 3](#_Toc90067547)

[2. Passing the Date to the other Fragments 3](#_Toc90067548)

[3. Displaying the Diary 3](#_Toc90067549)

[4. ROOM 4](#_Toc90067550)

[5. Navigation 4](#_Toc90067551)

[6. Extra Features 5](#_Toc90067552)

[4 The Code Implementation 6](#_Toc90067553)

[Main Activity 6](#_Toc90067554)

[SharedViewModel 7](#_Toc90067555)

[Entries 8](#_Toc90067556)

[EntriesDao 9](#_Toc90067557)

[AppDatabase 10](#_Toc90067558)

[DiaryRepository 11](#_Toc90067559)

[EntryViewModel 12](#_Toc90067560)

[AddFragment – Part 1 14](#_Toc90067561)

[AddFragment – Part 2 16](#_Toc90067562)

[AddFragment – XML 17](#_Toc90067563)

[DisplayFragment – Part 1 19](#_Toc90067564)

[DisplayFragment – Part 2 21](#_Toc90067565)

[DisplayFragment – XML 23](#_Toc90067566)

[app\_row – XML 24](#_Toc90067567)

[EntryAdapter 25](#_Toc90067568)

[delete\_menu – XML 27](#_Toc90067569)

[UpdateFragment – Part 1 28](#_Toc90067570)

[UpdateFragment – Part 2 30](#_Toc90067571)

[UpdateFragment – XML 32](#_Toc90067572)

[DateFragment 34](#_Toc90067573)

[FragmentDate – XML 36](#_Toc90067574)

[Navigation – XML 37](#_Toc90067575)

[5 Design 39](#_Toc90067576)

[6 Testing 40](#_Toc90067577)

[7 Evaluation 42](#_Toc90067578)

[Reflections On Implemented System 42](#_Toc90067579)

[Design Choices 42](#_Toc90067580)

[Personal Evaluation 43](#_Toc90067581)

[8 References 44](#_Toc90067582)

# Appendix of Code

[Figure 1Main Activity 6](#_Toc90067583)

[Figure 2Shared View Model 7](#_Toc90067584)

[Figure 3Entries 8](#_Toc90067585)

[Figure 4EntriesDao 9](#_Toc90067586)

[Figure 5AppDatabase 10](#_Toc90067587)

[Figure 6DiaryRepository 11](#_Toc90067588)

[Figure 7EntryViewModel 12](#_Toc90067589)

[Figure 8AddFragment-Part1 14](#_Toc90067590)

[Figure 9AddFragment-Part2 16](#_Toc90067591)

[Figure 10AddFragment-XML 17](#_Toc90067592)

[Figure 11DisplayFragment-Part1 19](#_Toc90067593)

[Figure 12DisplayFragment-Part 2 21](#_Toc90067594)

[Figure 13DisplayFragment-XML 23](#_Toc90067595)

[Figure 14app\_row-XML 24](#_Toc90067596)

[Figure 15EntryAdapter 25](#_Toc90067597)

[Figure 16delete\_menu-XML 27](#_Toc90067598)

[Figure 17updateFragment-Part1 28](#_Toc90067599)

[Figure 18UpdateFragment-Part2 30](#_Toc90067600)

[Figure 19UpdateFragment-XML 32](#_Toc90067601)

[Figure 20DateFragment 34](#_Toc90067602)

[Figure 21FragmentDate-XML 36](#_Toc90067603)

[Figure 22Navigation-XML 37](#_Toc90067604)

# The Process

## Introduction

This project when through a large variety of different versions throughout its creation. To start off the project I used Practical 5 as a foundation for the project code. This provided to be a relatively simple way of having 3 fragments, using a View Pager.

## 1. Storing the Entries - Database

The first thing I started to look at in the project was storing the entries for the diary. This is the part of the project that could the most considerable amount of time to work on. I initially started to look at preferences to store the date value, however these also proved to be a considerable challenge as they had limited features in how you can sort, search, or delete. Implementing any of these features manually seemed to be a bigger pain than was worth.

I then started using the database function that was demonstrated in the lectures. The integration of the database was relatively simple at first and I had it working where I could enter the entries into the first fragment and then view them within the Android files. To get them working I had to start looking at passing the data between the fragments since I couldn’t add an entry until I had both the date and the entry being added.

## 2. Passing the Date to the other Fragments

I started to look at intents and bundles to pass data to the entry fragment but spent far too long on it and could not figure out how to get it working. I scrapped this and eventually found out about View Models. This seemed to be an efficient way of storing values, but I ended up implementing singe View Models instead of shared view models and could not figure out where I was going wrong. Once I had the shared view models working, they were robust and kept this class until the end of the project. However, observing the view models and trying to pass the value out within other classes became a challenge as they only existed as local variables within the function. I, instead, had to create a text view for each page that would receive the input from the shared view model and then use that to pass it onto another variable within the fragment.

## 3. Displaying the Diary

Once I had the view models in place is where I started to run into issues that could most of time on the project. Once I had figured out how to make two fragments communicate (via view models) I then had the challenge of displaying the contents within the diary. My main issue with the database at this point was trying to have two separate fragments (add and display) accessing the same database. Many problems started to arise like how they were in fact creating two separate database instances, meaning that anything I wrote into the first fragment could not be read by the second and vice versa. I went through many iterations of trying to fix this problem like using companion objects as they are supposed to ensure that an object instance will not change but I soon realised this was not going to fix the problem.

This eventually led me to the discovery of creating a database using Room. Room is part of Android Jetpack project and has many built in function (e.g., sort, search, delete, etc) and this was highly recommended compared to using the SQLite APIs directly.

## 4. ROOM

Although very structurally beneficial to the project, Room took a considerable amount of time to learn as many of the tutorials and guides were written in Java. I started the project by creating the database and implementing a Data Access Object. A DAO is a database object which holds methods and properties that allow me to manipulate the room database freely by using its built-in properties. This made the creation much simpler and later deletion/updating entries far easier.

I then created a model of entries using the database which held the table within the database. This contained a primary key, the date of entries a long with the abstracts.

Through the Dao I then created a Diary Repository which abstracts access to multiple data sources and acted as a gateway to manipulate the database.

After a long period of testing and implementation the Room database seemed to be an efficient way of implementing a database using fragments.

## 5. Navigation

I was unhappy with how the page viewers and toolbar operated in my project. The toolbar meant that you could skip certain steps like selected the date which would cause an issue if you tried to add the entry without it and that could add the same entry by pressing the button multiple times. I also was not happy with the appearance of the toolbar, so I started to look at alternatives. I eventually came across the Navigation project which would let me lay out the fragments in a neat order and move the user onto the next page once they have completed an action. This implementation was relatively simple and hosted a few other features with it.

Calendar

Description automatically generated

## 6. Extra Features

I had now implemented the database through Room and had a navigation feature at my disposal. The next feature I would implement is the update feature which was initially designed to be a button that you click on to go to a different page, but I eventually thought it would be more intuitive if the user clicked on the entries and it would take you to the update page.

Graphical user interface, text, application

Description automatically generated

I also implemented the delete function along with this using an action bar from the navigation project. This meant that a symbol would appear at the top and the user would be able to delete the entry from there.

Calendar

Description automatically generatedI then implemented a delete all feature and then changed how the user selects the date. Instead of inputting values (as they user could choose any string they want) I instead decided to use a calendar function that would have an observer object that would receive any dates that the user selected and then add them to the entry once the user selected “add date”.

# The Code Implementation

## Main Activity

Figure 1Main Activity



The project starts off with the main activity which runs when the program starts. “binding” is a variable that stores the instance of the view binding of the Main Activity. Upon its creation, within the onCreate method, which instantiates a layout XML file into the Main Activity View. It then sets the activity content to an explicit view.

The *onSupportNavigateUp* is a work in progress method that was used to allow the user to use a ‘back’ button to navigate between fragments. This was scrapped due to the action bar causing the program to crash on start up. This is a deprecated method.

## SharedViewModel

Figure 2Shared View Model



This class extends the view model class and is used to pass the date that the user has selected between the *DateFragment* and the *AddFragment*.

The class holds a variable as a mutable live data of type string. Live data is a data holder class that is observable and is lifecycle aware. This means that it respects the lifecycle of other activities and fragments. This ensures that this data only updates app component observers that are in an active lifecycle. ‘Mutable’ means that it cannot be changed. So *\_data* is variable that cannot be changed and only updates with active components.

This is then stored as part of a live data that is changeable as we see in the *submitDate*method which calls non-mutable variable to have its value changed depending on the parameter value.

## Entries

Figure 3Entries



To store the diary data we need to define the entity that represents the object being stored in the database. The entity represents a row in the diary database. This includes:

* A Primary key – a randomly generate ID for each entry.
* Date – A string that holds the date of each entry.
* Abstract\_ - A string that holds the entered text of each entry.

## EntriesDao

Figure 4EntriesDao



The Data Access Object is an interface that contains the methods used for accessing the room database. This is a much more modular and efficient way of accessing the diary persisted data than using query builders or direct queries like you would have if you were using preferences.

The DAO provides many queries, such as:

* Insert – Generated implementation that inserts all parameters into the database. Ignores any conflicts in names/dates (so we can have multiple entries of the same date).
* Delete – Generated implementation that removes a set of entities from the database, using the primary key.
* Query – Generated implementation that passes an entity through a parameter to access a given subset of the database. In this case we use it to query the entire database to delete or display it.
* Update – Generated implementation that changes a set of entities in the database, using the primary key.

## AppDatabase

Figure 5AppDatabase



This class represents the database in the Room library that serves as the main access point for the underlying connection the diary’s persisted, relational data.

This class extends the Room database class and holds a reference to the DAO. The entities are identified as “entries” and a single instance of the diary database is instantiated.

Within the *getDatabase*class, it checks if the instance already exists in which case it is synchronised with the class. If the instance is not detected then a new instance is created by calling the *databaseBuilder*function from the Room project. The application context and the *AppDatabase* class are passed as parameters to create the instance.

## DiaryRepository

Figure 6DiaryRepository



This class is a repository abstraction that is used to hide the multiple data sources we use in our diary (e.g. internal database). This repository holds the destination to the data sources from where to retrieve information about the Room database. It interacts with the DAO by passing it through its parameter and accessing its methods.

The repository is not necessarily part of the Architecture Components libraries, but is a suggested best practice for code separation and architecture.

## EntryViewModel

Figure 7EntryViewModel



This class acts as the View Model between for the diary. A view model provides data to the UI that survives configuration changes e.g., screen rotation. It also acts as a communication centre between the Repository and the UI.

This class extends the Android View Model of the application and holds access to the database and repositories.

Upon initialisation, it instantiates the DAO for the application and the repository. It uses the repository to access all the entries from the *Entries* class and stores them as a live data list of entries (table).

It then provides kotlin coroutine methods as accessors to the repository class. This means that the methods are called as coroutines which run as threads in the background. This is useful for methods that execute asynchronously and that would otherwise block the main thread and cause the app to become unresponsive. This allows for more efficient and robust applications.

*submitDate, \_date* and *date* are deprecated duplicates from the other view model class.

## AddFragment – Part 1

Figure 8AddFragment-Part1



This fragment is the third page that appears in the sequence of the program. This page allows the user to enter their text to be inserted into the diary.

Upon creation, this class holds instances of both the date and entry shared view classes. This is so that it can gather the date set by the user in the data fragment and make changes to the database class through its repository.

To retrieve and store the date from another fragment, it calls upon a observe method on the live data variable stored in the *SharedFragmentView* and stores the observed value into a text view called *dateView*.

Both *btnAdd* and *btnCancel* call the methods insert diary entry and cancel respectively.

## AddFragment – Part 2

Figure 9AddFragment-Part2

A method is created to add the date and entry values to the database. Date is taken from the text view as mentioned earlier and the abstract is taken from the user’s input into the *editText*. It then calls a method to perform an input to check that the user has not left the entry field blank and stores the id, date and extract as an *Entries* object. This then gets passed to the *addEntry* field within the shared model which will eventually go through the repositor, dao and then to the database. It then displays a success message to the user and takes them back to the home page. If the user has left the field blank, then the entry is not added, and the user is given a message prompt telling them they still need to fill it in and they remain on this page until they do so. This is the method called when the user clicks the button to add the entry.

The method created to check the user’s input simply passes the date and abstract as parameters and uses *TextUtils* feature they are not empty.

The method to handle when the user clicks the cancel button is created by simply displaying a message to the user telling them have cancelled and takes them back to the home page without adding any entries. Any entries that were not added are not saved when cancel is selected.

## AddFragment – XML

Figure 10AddFragment-XML



Graphical user interface, text, application, Teams

Description automatically generatedThe *fragment­\_add* XML is a constraint layout that contains two buttons, a text view, and an edit text.

* *dateView* – is a text view (non-editable) which is used to hold and display the value taken from the date fragment. This is purely for display and storage and not meant for the user to interact with.
* *editExtract* – is an edit text which receives a multiline string input, meaning that the user can write lines of string and not have it display as one continuous line when they are typing it. Otherwise, when they pressed enter for a new line it would submit the entry and move back to the home page.
* *Btn\_add* – Button clicked by the user to access the methods to add the *editExtract* text to the database and move pages.
* *Btn\_cancel* – Button clicked by the user to cancel their current data entry and move back to the home page.

## DisplayFragment – Part 1

Figure 11DisplayFragment-Part1



The display fragment is the initial and home page that the user will be greeted with when they open the application for the first time. From here, the user can view and modify any current entries, as well as delete all entries from the action bar.

Upon creation of the class, the view model class for the database is accessed and an instance of the *EntryAdapter* class is created (see **EntryAdapter**). It uses the adapter instance and stores it as the value for the recycler view’s adapter (see **DisplayFragment – XML**).

Like other fragments, it holds a reference to the *EntryViewModel* which holds the access to the database methods. It calls the observe function on the *getAll* variable within the view model. This variable is the live data value of the list of *Entries* – providing us with all entries within the database. It then passes this variable through the *setData* method within the *EntryAdapter* instance (see **EntryAdapter**) so that each entry is displayed on the main page.

*DisplayFragment* holds a button to allow the user to add a new entry. The Navigation class method *findNavController* is used to navigate the user to the date page.

The fragment then adds the functionality for a menu at the top of the screen, this will hold the delete functionality. This is provided by the *androidx.fragment.app.Fragment* package.

## DisplayFragment – Part 2

Figure 12DisplayFragment-Part 2

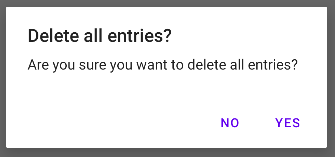


Along with the view binder, it requires an override method on *onDestroyView* which handles when the view binder is detached from the fragment and so the inflater for the fragment is set to null. The next time the fragment needs to be displayed, a new view will be created.

The menu function taken from the *androidx* fragments directory holds an *onCreateOptionsMenu* which is overridden and calls a menu object called *delete\_menu* (see **delete\_menu**) and inflates the layout for it.

This object has another override function *onOptionsItemSelected* which is called when an item is clicked on the menu. In this case it checks if the icon clicked was the delete menu and calls the function to delete all entries in the database.

The function to delete all the entries starts off with an AlertDialog which looks like this:



As seen above, the dialogue box asks the user if they are sure about deleting all entries and provides two buttons for the user to select. If the user selects “No” then the dialogue closes and they return to the page they were on. If the user selects “Yes”, however, the *deleteAll* function from the shared view model is called (see **EntryViewModel**). It also provides a success message for the user and closes the dialog box.

The entries within the database work on a shared view model structure so the display fragment should update instantly revealing that all entries have been removed. Otherwise, the methods would have to be called again to read the entries from the database and/or the user would have to be navigated to a different page to refresh the page.

## DisplayFragment – XML

Figure 13DisplayFragment-XML



A picture containing graphical user interface

Description automatically generatedAs mentioned previously, the display fragment page contains a recycler view (see **EntryAdapter**) and an add button.

The add button is a custom icon “*ic\_add*” taken from the Android Studio clip art that navigates the user to the date page and is displayed in the bottom right corner.

I used a recycler view instead of a simple text view for several reasons:

* Recycler views can make use of customizable item layouts like I have used in *app\_row* (see **app\_row - XML**). This allows me to keep the display as a single object and hold separate custom objects instead of many separate view texts.
* Recycler view has amore extensible framework compared to ListView or GridView especially since it provides features such as search (which could be implemented in the future) and horizontal and vertical layouts.

## app\_row – XML

Figure 14app\_row-XML



Text

Description automatically generated with low confidenceAs discussed previously, this layout design is for the customizable object that is displayed as rows on the display page.

It is a simple layout with a text view at the top of the entry displaying the date as a string and the diary extract in the text view below. The extract text view will hold a vertical scroll bar that can be used by the user if the extract is longer than the container object.

## EntryAdapter

Figure 15EntryAdapter



Recycler views require that they are implemented with an attached adapter to handle the data being passed to it. *EntryAdapter* acts as the adapter for the recycler view within *DisplayFragment*.

This class extends the recycler view adapter class which passes our inner class *MyViewHolder* as the view holder.

The inner class *MyViewHolder* extends the recycler-view view holder class. This view holder class provides a direct reference to each of our views within the recycler view. The view holder is also used to cache the views that will be set as the render a row. This class passes through the view binding for the custom row layout to use as a custom row in the recycler view.

The *onCreateViewHolder* passes a *ViewGroup* and an integer. *ViewGroup* is a special view that contains other views (children). This acts as the base class for layouts and view containers and is used to pass the context to the view holder.

The *onBindViewHolder* is overridden to display the data at the specified position in the recycler view. It passes in the view holder object the position of the entry within the database as an integer. It stores the current entry in a variable and retrieves the current item’s date and extract value from the *app\_row* custom row.

Allows the *diaryView* text to be scrollable. This incorporates the scroll bar and allows the user to drag the extract to scroll up and down the container to reveal information that was otherwise hidden.

The view holder then listens for any clicks on any of the custom row layouts on the display page. If the user clicks an entry, then the view holder will call the navigation method from the display fragment to the update fragment which is taken from the display fragment view binding.

Adapter holds getter and setter methods for number of entries and change the entry table to the current one, respectively.

## delete\_menu – XML

Figure 16delete\_menu-XML



Shape, rectangle

Description automatically generatedSimple action bar that appears at the top of the display page. Holds a custom symbol button as *ic\_delete* that produces the dialogue asking the user if they are sure they want to delete all entries. Shows the button if there is room for it (*showAsAction*).

## UpdateFragment – Part 1

Figure 17updateFragment-Part1



This is the fragment called when the user clicks on one of the entries in the display fragment. This is very similar to the *AddFragment* as it starts off with a reference to the entry view model to modify the entry database. It allows the user to modify a currently existing entry in the database or delete the selected entry.

Upon creation, it populates the two edit texts in the page with the date and diary extract that were chosen.

Using the navigation project, the *Entries* class is passed as an argument called “currentEntry” of type navigation argument. This is stored as a variable that is used to access the selected diary and date fields from *Entries*.

The cancel button calls the same methods as before (see **AddFragment – Part 1**) and creates the same action bar at the top of the page (see **DisplayFragment – Part 2**).

The user can only modify the entry text and not the date. This prevents users from having to manually insert the date in a string format and keeps the order of the rows consistent. This also follows a diary more accurately as they do not often let you change the date – this is done automatically.

## UpdateFragment – Part 2

Figure 18UpdateFragment-Part2



The rest of this class is basically identical to the *AddFragment* class as they use the same principles.

The main difference is that when we add entries to the database we take the existing ID and date and pass them through the *updateEntries* in the entry view model along with the new text entry.

There is also the addition of the action bar at the top of the page that contains the same programming as the one in *DisplayFragment* with the exception that it calls the method to delete the current entry and not all of them. The same dialogue box and options are displayed when clicked.

## UpdateFragment – XML

Figure 19UpdateFragment-XML



Graphical user interface, text, application, email

Description automatically generatedExactly the same layout as *AddFragment* (see **AddFragment – XML**) except that there is an “Update Entry” button instead of “Add Entry”.

## DateFragment

Figure 20DateFragment



This is the fragment that the user navigates to when they have clicked on the “add” button on the home page. This fragment prompts the user to select a day from a calendar view, stores that value as a string in a shared view model and then navigates the user to the add fragment page.

The class instantiates with a reference to the shared view model that stores the date variable.

Upon creation of the class, it is listening to the calendar view object and noting any changes the user makes to the calendar. Once the user makes a change, the chosen date is stored as a string in a text view and a message displays telling the user the current date they have picked.

Once the user clicks the “Apply Date” button, the class will read the currently selected date from the text view and pass it through the shared view model reference to call the *submitDate* function with *SharedViewModel*. The class then navigates the user to the add fragment page.

## FragmentDate – XML

Figure 21FragmentDate-XML



Calendar

Description automatically generatedAs you can see by the position of the text view, it is not necessarily meant for the user to see but rather to store the string version of the date the user has selected.

This layout contains a simple Calendar view provide by Android and a button to submit the date.

If the user decides they want to cancel they will have to navigate to the next page to do so.

## Navigation – XML

Figure 22Navigation-XML



Graphical user interface, application, Teams

Description automatically generatedContains all the navigation tools required to move between fragments. Display Fragment is used as the host where the user will begin at the start of the program. From here they can choose to either add or update an entry where they will be directed to the date or update fragments respectively.

Update is inaccessible outside of the display as well as the Date fragment as these require you to take action to change/add entries.

Update and Add will always point back to the Display to show the user the changes they have made to the entries.

# Design

Diagram

Description automatically generated

# Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case Description | Fragment | Input Data Requirements | Expected Results | Pass/Fail |
| The Program starts when pressing ‘run’ | - | - | Program starts at the Display screen | Pass |
| Pressing the plus button on Display page | Display | User clicks | Navigate to the Date page | Pass |
| Selected a date | Date | User selects 01/21/2021 | Navigate to Add page.  Date is displayed at top of screen | Pass |
| Navigate to modify entry | Display | User clicks on an entry | Navigate to Modify | Pass |
| Modify entry | Update | User adds “Hello, this is a test” to extract | Navigate to Display  Entry is changed to hold different text | Pass |
| Delete all entry dialogue | Display | Click the delete icon | Dialogue box will pop up asking if the user is sure | Pass |
| Select “No” dialogue | Display | Click “No” | Dialogue box will close | Pass |
| Select “Yes” dialogue | Display | Click “Yes” | Dialogue box will close.  All entries will be deleted.  A message will display telling the user they have been deleted | Pass |
| Delete all entries when there are none | Display | Click delete icon and press “Yes” | Continue the same as if there were entries. | Pass |
| Select a date in the future | Date | Click 01/12/2030 | Work as normal. Diary entries should be able to be made for the future | Pass |
| Select a date far in the past | Date | Click 23/10/1900 | Work as normal. | Pass |
| Cancel diary entry | Add | Click “Cancel” | Navigate to display page.  No entry/date should be stored. | Pass |
| Keep the current date as the same and not interact with the calendar | Date/Add | Click +  Click “Apply Date”  Click “Add Entry” | Date should be stored as the current day | Fail – No date is entered, and date is stored as “TextView” |
| Media back button on emulator for each page. | Display/Date/Add | Navigate to Add and then press the “back” button twice | Navigate back to Display.  No entry added | Pass |
| Media back button after inputting an update | Update | Click on entry.  Type text in extract field.  Press the back button | Navigate back to display.  No changes are made on the entries. | Pass |
| Cancel update | Update | Click on entry.  Click cancel. | Navigate back to display.  No changes are made on the entries | Pass |
| Update field with no text in extract | Update | Click on entry.  Delete text.  Click “Update” | Warning displayed to add entries to fields.  Stay on Update | Pass |
| Try and edit the date in Update | Update | Click on entry.  Modify Date  Click “Update” | Date should not change. | Pass |

# Evaluation

## Reflections On Implemented System

This project had an incredibly slow start due to personal events and general implementation and design problems with Android. At the start of the project, I had a limited understanding of how I was to implement any of the required features or how it was going to look in the end.

However overall, I am very satisfied with the way that the project turned out as it manages follow the basic design structure that it started with but in a way that is much more responsive and fluid. There were many design changes but the one I came up with in the end has produced a relatively good piece of work.

The navigation system seemed to have been a big change in the layout of the program as getting rid of the tab layouts was a wise decision in the prospects of the project. It implements many Android features, such as:

* Kotlin coroutines to help prevent the main thread from getting filled up with processes.
* View models that seamlessly update with every change that the user makes to the database.
* A well-structured database access system using repositories and a DAO.
* A navigation controller that makes traversing the application so much simpler and intuitive within the pages.
* An action bar with icons and dialogue boxes.
* An extra update feature that is intuitively accessible from the display screen.
* A calendar view that lets the user select a date from a calendar, making it more accessible.

There are still several features, however, that I would have hoped to include but could not due to time constraints:

* Search function for the date.
* A more design-friendly interface.
* An option to sort the entries by recently added/date.
* A main menu.
* Support for multiple devices.

## Design Choices

There are a few design choices that I made in relation to the Android Design Principles [1]. The icons used in the display screen and the action bar were carefully chosen to be clearly visible and identifiable to most users. The use of a floating action button is to encourage user action [2] and is used to represent a key feature within the application. Many users see it as a “wayfinding” tool [3] that when faced with an unfamiliar page they will look to use a floating button for navigation.

The action bar I used in this project is a suitable design choice as they are most appropriate for applications with a relatively few number of options [2]. However, the action bar is out with thumb zone which means that it may be more difficult to delete entries if you are only using one hand – which is the case for roughly 49% of users [4]. This is less of a problem if you are using two to type.

What I had not considered when implementing the phone is how the size of the buttons and fields would translate to a physical phone rather than an emulator. I had picked pixel sizes for the buttons and fields based on what would appear on an emulator that turn out to be too small to be optimal sizes for users. It is recommended for medium priority buttons that they be roughly 60 pixels [5] whereas the buttons used in this project are between 40-50. This may cause some difficulty for some users when selected between fields that are close together, such as the entry field and the submit button.

There are also little to distinguish between entries on the display page. Many diary applications include a design feature that allows the user to quickly distinguish the separate entries. In the case of this application, I have used space for simplicity but in hindsight does not look appealing as other design features and may be confusing for smaller entries.

The application also has a very limited colour palette due to time constraints which makes the application as a whole look blander and more unappealing to use. Considering the simplicity of the application, what distinguishes one diary application from another is its design layout and appearance which I think is considerably lacking within this application.

## Personal Evaluation

Throughout this application I have been thorough in my design and implementation to ensure that I create an application that I am pleased with. I think I have managed this to an extent, and I am happy with the outcome. Due to personal setbacks, I had a considerable delay to the start of the project. Once I had started the project and understood the scope of it, I did not think that this project would be submitted as a working piece of work due to the amount of time I had left. However, managing my time efficiently and becoming more skilled at self-taught coding I have managed to create more than I was expecting.

The concept of fragments and activities was very alien compared to other projects I have worked on so that was the biggest barrier I had to overcome in completing the project. I have finished the project, however, with a much greater understanding of kotlin and android systems.

If I were to do this assignment differently, I would have chosen to priorities my time very carefully and scrutinise myself to stick to the essential features of the program and not bother with the more advanced features. This would ensure that I got the assignment in in a more reasonable time, granted with less features and marks.

# References

|  |  |
| --- | --- |
| [1] | Android Developers, “Android Developers,” Android, [Online]. Available: http://android-doc.github.io/design/get-started/principles.html. [Accessed 7 December 2021]. |
| [2] | N. Babich, “Adobe,” Adobe, [Online]. Available: https://xd.adobe.com/ideas/principles/app-design/essential-patterns-mobile-navigation/. [Accessed 8 December 2021]. |
| [3] | B. Fong, D. Chiu and Z. Gibson, “Google User Research,” 26 June 2014. [Online]. Available: https://youtu.be/dZqzz5lZFvo?t=23m9s. [Accessed 8 December 2021]. |
| [4] | S. Hoober, “UXMatters,” 18 February 2013. [Online]. [Accessed 8 December 2021]. |
| [5] | Anthony, “Optimal Size and Spacing for Mobile Buttons,” 29 February 2019. [Online]. Available: https://uxmovement.com/mobile/optimal-size-and-spacing-for-mobile-buttons/. [Accessed 8 December 2021]. |