Ryan Mcgoff (4086944), Jaydin Mcmullan (9702973), Jerry Kumar (3821971)

FlexiTask

What we learned from assignment 3

From David's comments for assignment 3, we took on board three important pieces of information we wanted to work on.

1) Design. We really wanted to improve the design of our app, this included adding our own basic scaling animations, custom toolbar animations, and employing a consistent color scheme throughout.
2) Testing. Manual testing is still a big part of our engineering process, due to the complicated nature of unit testing and our unfamiliarity with it. However for assignment 4 we improved our limited understanding by designing a couple of unit tests that would test methods we had used in our app, we attempted instrumental testing, but ran into issues when it came to creating mock contexts that required an advanced understanding of Android.
3) Comments. We learned in assignment 3 we had been using JavaDocs incorrectly. Comments with // or /* are ignored by the compiler and some of our classes and method headers used these comments. For assignment 4 we made sure we were using the correct style and that the comments were easy to read and understand.

Planning & Team Communication

Just like with previous assignments, we used a shared Google Sheets document to indicate the days we were free, highlighting the days we were not in red. We aimed for at least three days a week to meet up and work on the assignment together, planning which parts of the app we wanted to work for that day. We had 6 weeks to do the assignment in, so wrote a rough outline of each task that needed to be completed, with an attached difficulty/time rating, and aimed for ten points per week.

Java
Labels (10)
Animations/design(5)
Fixed Task alarms(10)
NotificationHelperReminder(10)

XML
Dialog xml files (10)

Other
Colour deficiency options(5)
Comments(5)
Bugs(5)

We all worked together and at the same time, but for each major implementation we had a seperate team lead, whose responsibility it was to research the feature and construct a plan on how we would tackle it. Ryan worked on xml design/animations, Jerry on notifications and Jaydin on labels. When it came time to work on a particular feature, the team leads would present their plan and share any knowledge they had learned in their research endeavors. This way we could spend our limited time together working on our app, rather than reading android documentation as at least one group member would have experience working with that feature.

New Additions

Animations were a new addition for the final assignment. We created a toolbar animation using simple scalers and timers to convey the effect of the toolbar loading in first and the checkbox, edit and delete symbol loading in afterwards. This gives the app a more playful feel and makes the process of interacting with a task slightly more satisfying.

Touching the newly designed header in the navigation draw takes you to the productivity activity. This activity lets you set a weekly goal of tasks you would like to complete and displays that progress in a filled circle (created using a third party animation library), along with lifetime stats of completed tasks. To make this information even more readily available to users, we

used a view switcher in the navigation header to switch between displaying these stats in a text view - this lets the user view these stats from the main activity.

Labels can now be created in the navigation draw, and added to any task. The user can then filter tasks by label. Labels are stored in their own SQL table for efficiency.

Separate alarms for fixed tasks can now be added if a user chooses. The user can set as many of these as they like, choosing what time before the task is due they would like to be reminded.

For the new app logo we decided to go with a combination of the letter F (flexitask is the name of our app) and check mark symbols(typically associated with task managers). We aimed to create an original icon that would adhere to material design principles and incorporate the colors used in our app. None of us had any experience using photoshop, so we keeped the design simple.

Disability Support

We've really tried to focus on enabling users with colour deficiencies to utilise our app just as effectively as users without those disabilities. Along with supporting content descriptors for screen readers and large touch targets for visual impairments (which we tested with a visually impaired group member), we have custom built a color appearance setting for colour blind users. We currently support three of the most common forms of color blindness, deuteranomaly, protanopia and tritanopia - with a framework in place that would make it easy to add more in the future. While android phones do come equipped with system level options for color blindness, they don't guarantee the color the system will choose will look nice. What looks good to a non-color blind user might look terrible to a color blind user.

We spoke with Andrew during assignment 1 when we were struggling to think of disabilities to support that weren't already supported at a system level, he suggested memory impairment. To support this we implemented a history page showing users of their previously completed tasks in the past year, they can use our filters to filter down to a more specific time period (ie: tasks completed today) and by task type. We also have two types of notifications. Daily summaries can be turned on in the apps settings and will send users a daily notification showing them 10 upcoming (and overdue) tasks in the next week. Individual reminders, are optional reminders

that can be set to remind the user of a task before it is due. For instance, maybe you want to be reminded of a birthday party 1 week before it begins.

General Testing & Accessibility Testing

We continued to employ a manual sandbox method for testing, this allowed us to isolate some of our app's functionality into separate apps for testing. We used this to determine the best way to communicate between a fragment and a navigation drawer in the parent's activity for our labels system (we settled for a local broadcaster, see next section for more information). And also made a separate app to test tangqi92's wave library, before we began implementing it into our final app.

To test how our UI would respond to accessibility changes, including text size changes, we used a group member who suffers from minor visual impairment. They said they were satisfied with how the app responded to their phone's accessibility settings. In addition to this, we tested to make sure how app would work well with screens of all sizes, including large tablets and smaller old phones.

We used Colour Hexa to discover colours that were generally opposite each other on the colour wheel from a selection of colours that would be visible to colorblind users. We then used a colour blindness simulator to make sure the colors were suitable, once we were confident, we tested our app on a friend with Deuteranomaly and a family member with very mild Protanopia. Most of this testing was done for assignment 3, so for assignment 4 it was just a matter of responding to their feedback and changing the appropriate color values in the resource folder.

In addition to these tests, we also implemented three unit tests that were copies of methods our app used. We tested our Flexi task's priority checker, if our app was correctly labeling a task as due/overdue, and if flexitask was setting the reminder with the appropriate values. In an ideal situation we would have used instrumental tests instead, but creating a mock app context was incredibly challenging and time consuming, so we just implemented standard unit tests as a compromise. (SEE FILE FlexiTaskUnitTests.java for the implementation)

We tested the device on as many apis and screen sizes as we could, to make sure all the text was visible and none of the support libraries we used would crash those devices.

Current known issues in the final release:

- When the user rotates the screen, the UI resets itself. This is a lifecycle problem with our app. Life cycle's + saved instances are incredibly challenging to work with and are beyond our current understanding of Android, especially with a viewpager, container and a navigation bar in the mix. We've done our best to manage this, making sure no user input is lost on a device rotation, however if you're in the history fragment or settings fragment and you rotate the device it will take you back to the main task page.

Problems we found and fixed

Getting information from one activity/fragment to another activity/fragment was a big challenge. For instance when a user indicated they wanted to delete a label, we needed to communicate back to main activity so the main activity could remove the label from the navigation draw. Broadcast receivers required less overhead than an interface, so we used this. For our dialog fragments (like "Create new label" or "Edit Weekly Goal"), we used a interface and a listener to communicate with the active class and notify it of when a user had made a change to the input.

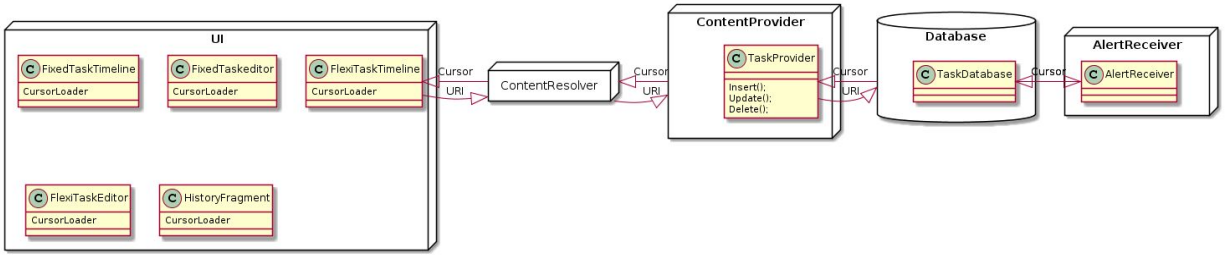
Another challenge we solved was the issue that occurs when a device is turned off, all the set alarms are destroyed in this process. This meant that with our assignment 3 implementation of notifications, all scheduled notifications would be destroyed when the device was off. To combat this we used a special Android system level intent, to notify our app that a user's device has turned on again. This intent called our DeviceReboot class and set all the alarms that have been switched off, checking to make sure the due date hasn't already passed before setting them.

Version control

We ran into issues with Android studio not accepting our old git repository from assignment 3. We thought it might be down to credential errors, similar to what we experienced in assignment 2, but it wasn't. After spending two days trying to fix it, we decided to cut our losses and start a new git repository so we could get on with the coding. While not an ideal situation, we still had our old repository on GitHub, so any changes we wanted to revert back to we could. It just made

the process a manual task rather than a automatic one. Throughout our final release we would make sure to commit and push to the master.

How the app works



Our mainactivity (mainActivity) acts as a container to hold and switch fragments, based on what item the user has selected in its navigation draw. Our Timeline Fragment Container is selected by default and contains a viewpager containing the fragments for Fixed Timeline and Flexi Timeline, this viewpager allows the user to swipe between each timeline fragment. When these fragment classes are first created (onCreate() method is called) each class initializes its own loadermanager. Once this loadermanager is up and running, it calls the OnCreateLoader() method, which initializes a new CursorLoader. This cursor loader is given a projection (a string array of the columns the activity/fragment want), the URI (like a URL but for database tables) and a sortOrder (Flexitask wants the data returned to be sorted by its priority calculation, whereas fixed task it wants the data ordered by due date). This cursorloader in turn queries the contentresolver.

The contentresolver “resolves” the URI to our custom content Provider (Task Provider). This provider acts as an interface to the database, it’s here that determines whether we are inserting, updating, deleting or just querying data - and whether we want a specific row or the entire table. It performs the desired task and returns a Cursor with the data requested, back to the contentResolver, which returns back to the LoaderManager which passes it on to the cursorOnFinish() method. In this method we call the cursorAdapter swapCursor() method and pass it this newly returned cursor to process.

By using the loader thread and abstracting this data retrieval process to loaders and content providers, we allow the user to continue using the UI with few interruptions. We also use a similar querying technique in our editor activities, in addition to a simple if() statement that

allows us to reuse the editor activities for both creation of a new task and updating an existing task.

We had to account for when the user “updates” the task from either the flexi or fixed timeline fragment in the form of the “done” button. When selecting “done” (the tick), for a fixed task, the app will check if that task is recurring or not. If it is it will update the tasks due date to (the current due date + recurring period * 86400000 (milliseconds in a day)) to get the new due date. It then restarts the cursor loader to go and retrieve these new changes.

When a flexi task is first created the date-last-completed field for that row/task is set to the current date (in milliseconds) and the next due date(in milliseconds) is derived from this by adding the recurring period * milliseconds in a day. When the user indicates they have finished an activity (by pressing the done button/ tick), the date-last-completed field is set to the current time and the next due date is recalculated.Our priority rating is calculated at two stages, one in when we query the data in the Flexitask fragment loader and want the return data sorted by priority and the other in the cursoradapter where we assign a XML color element to indicate the priority to the user.

This is the algorithm for the priority:

$$\frac{(\text{Todays Date} - \text{Date Created})/86{,}400{,}000 + 1}{\text{Recurring Frequency}}$$

This produces a number which determine how overdue a task is. We then organise the flexi tasks based on which tasks have the larger number, as well as assign priority colors (based on the user’s colour deficiency settings).

The HistoryFragment works much the same as the flexi and fixed task timelines, in that it uses a loader manager and cursor adaptor to obtain “deleted”(deactivated) tasks. The history fragment has several buttons and corresponding listeners, that act as filters, when a button is selected it restarts loader which retrieves the appropriate data using the new parameters.

When our app is started, our mainActivity creates a system alarm for a specified time (at first this is 8:00am, but can be changed in the settings). Once this alarm is triggered (eg: @8am),

the Android system calls our AlarmReceiver class which cursors through upcoming active tasks, and adds their titles to an array. Once it has obtained these tasks, it utilises our Notification Helper class to create a Notification. This class first creates notification channels for Android devices running Oreo or higher to use, and creates a message with the given array of task titles from Alert Receiver. The result is returned to alert receiver and broadcasted to the users device. Alert receiver then sets up an alarm for the following day.
Labels are created in the label dialog fragment, which lets the user enter a new label, and implements a listener interface to let main activity know the user has created a new label. The main activity then adds the label name to an SQL table and adds the label name to the navigation bar for the user to select. Upon selecting the label, the navigation bar, which resides in main activity, stores the selected label into a shared preferences database and resets the timeline fragments for flexi and fixed tasks. These timeline fragments onCreate(), check to see if this shared preference database is filled and change the SQL call to only return tasks with this label. The user can delete the label while in one of these timeline fragments, and when they do, the timeline uses a deletes the label from the SQL table and uses a local broadcast to tell mainactivity to remove that label from the navigation bar.

Where to from here

We've just uploaded our app to the app store, so our first priority will be responding to any feedback we receive. We've provided a way for the user to contact us in the app's settings. Android has a huge range of different devices and APIs, emulators are great, but they don't catch everything and we can't realistically test every device. After we have responded to the initial feedback and fixed any reported bugs, we plan on implementing a new type of task called Irregular tasks. These are tasks that don't have a fixed due date or a flexi-due date. These are simply activities you do occasionally completed that you would like to track how often you do it and the last time you completed it. For instance, you might want to know the last time you saw a friend, or the last time you changed your car's oil.

We're learned a lot during this assignment, not just in terms of Android development but also how to work well in a team environment where each member has a different set of priorities and time schedules. We managed to deliver on almost everything we promised in assignment 1.

How to build our app

1. Download and install the latest version of Android Studio with the default settings (https://developer.android.com/studio/)

2. Download our project from Github
3. Unzip the file
4. Import our project into Android Studio (File->Import) or select the “open an existing Android Studio project” from the startup menu
5. Then browse to the FlexTaskBeta-master folder, make sure this folder shows an Android symbol
6. Once it’s loaded (might take a while, as Gradle may need to download files), press the “RUN APP” button and select **Pixel** as the virtual device (the device we used for testing, but any device using an API higher than 19 should work) and click ok

NOTE:

- ***To test the different colour options, you must close and reopen the app, as the toolbar’s colour can’t be changed programmatically once inflated and requires a restart.***
- ***Notifications can be tested by setting a time in the settings (otherwise it should default to 8:00am) to be one minute ahead of the phone's settings.***
- ***The history section contains bugs at the moment due to how we’re processing the task data and only displays tasks that the user explicitly deletes.***

Google Play

Our app as of writing this is currently being reviewed by Google Play, it should be searchable by typing “flexitask” into Google Play’s search bar by the time you read this.

Thanks to

Code used:
https://github.com/tangqi92/WaveLoadingView (for wave animations in ProductivityActivity.java)
https://stackoverflow.com/questions/5533078/timepicker-in-preferencescreen/10608622 (for timepicker in Shared preferences)