Software Engineering 2
FINAL REPORT

Team number:	0206

Team member 1	
Name:	JIN YU
Student ID:	11717460
E-mail address:	a11717460@unet.univie.ac.at

Team member 2	
Name:	Merve Sen
Student ID:	01500082
E-mail address:	a01500082@unet.univie.ac.at

Team member 3	
Name:	Edna Mutapcic
Student ID:	11709017
E-mail address:	a11709017@unet.univie.ac.at

Team member 4	
Name:	Mohammad Mahdi Fallah
Student ID:	01428941
E-mail address:	mahdyfalah@gmail.com

Team member 5	
Name:	Sahel Naderi
Student ID:	11823360
E-mail address:	a11823360@unet.univie.ac.at

1 Final Design

1.1 Design Approach and Overview

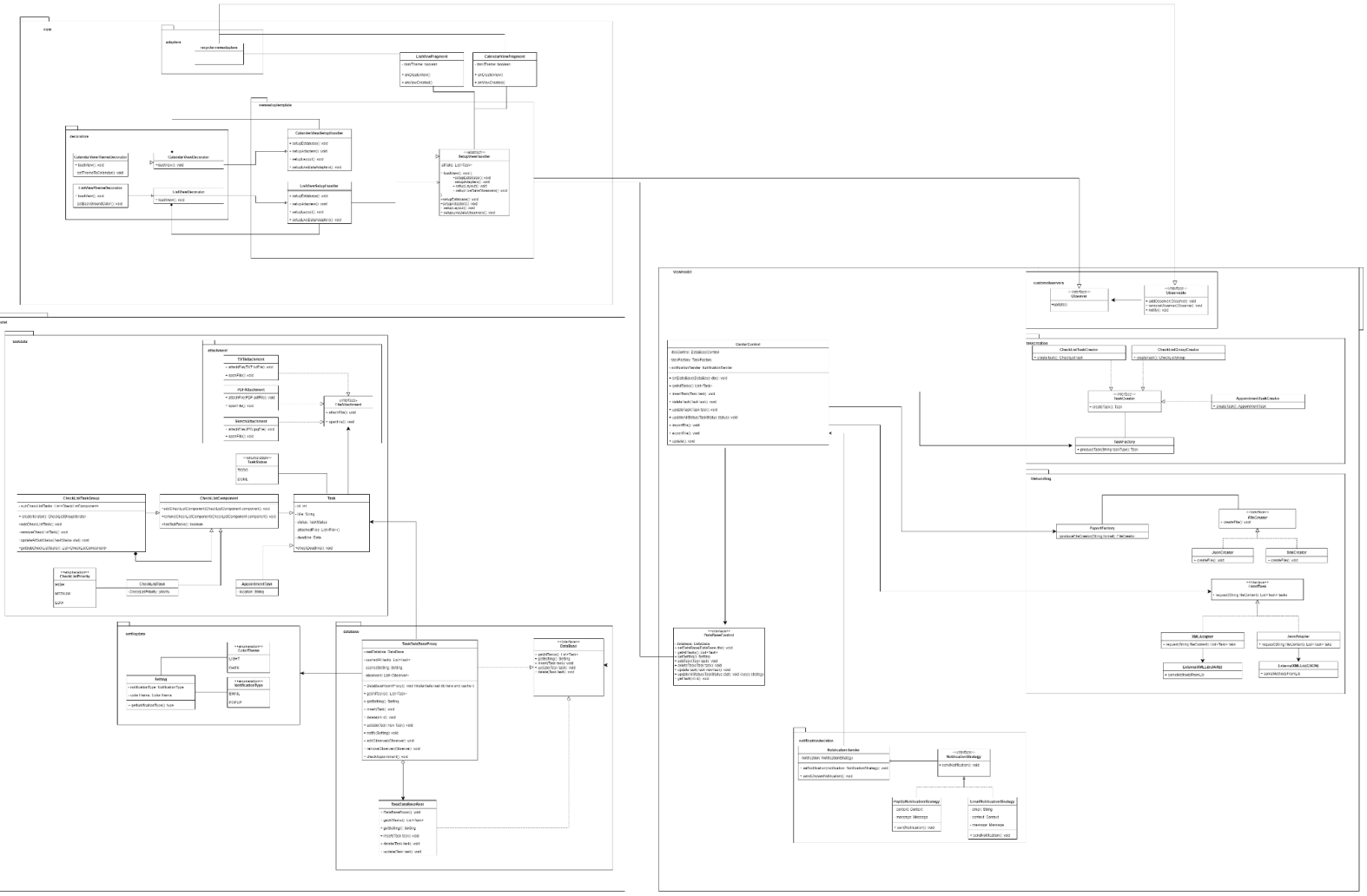


Figure 1.1: Entire UML for the whole project

The design approach on this phase focus more on individual work, which means that the implementation was done separately by each member as well as the iteration/changes on design along with UML. At the end all team members will explain their implementation and changes in a meeting before informal code review. (Note: the dbs implementation was done and was already fully functional on the first DESIGN phase, the implementation should be more independent and is less likely to be not blocked by other members.)

There are some minor changes, as we discovered more functionalities offered by Android as well as Room dbs (will be discussed in details in ***section 1.2***). Here will be a short summary for the general design approach separated by each members.

Member 1:

My approach for

- creating of different type of tasks: a Factory pattern will be used.
- inserting, updating, deleting and storing all different type of task: with Android Room database.
- updating the common properties: custom query defined in the Dao interfaces, along with the use of database (different that my initial plan during the DESIGN phase, will be discussed further in ***section 1.2***).
- displaying, adding, editing, deleting them in the UI: LiveData, RecyclerView, Proxy and Intent. In more detailed, LiveData and RecyclerView are used for displaying. Proxy is used for communicating between client and the database. Intent is used for passing data between activities (e.g. adding/editing a task need a new activity. Soon as the activity closed, the data in that activity (AddEditTaskActivity) will send all the data back to MainActivity using Intent).

Major packages under my responsibility:

- ***customobservers***: contains the interfaces for Observer pattern (pattern detail will be explained in ***section 1.3***)
- ***database***: contains all necessary classes for Room database. (including Dao interfaces, proxy pattern and database initialization, also a converter class for converting some complex datatypes to be easily stored in the dbs)
- ***recyclerviewadapters***: 3 different adapters (made for Appointment, CheckList, CheckListGroup task) for linking the data and “feeding” the recycler view.
- ***taskdata***: contains all classes related to different type of tasks, as well as a composite pattern to handle parent and child task.
- CenterControl and DataBaseControl: Two classes that are used to communication to the database.

Member 2

My responsibilities in this project were:

- That a task type should be composed of subtasks, where we defined the groups as checklisttaskgroups the subclasses as checklistcomponents.
- To find a suitable strategy for deleting tasks from the main and/or subtasks, in which we decided to use the Composite Pattern (more details will follow).
- Each task type should support at least two file format attachments, whereby we provide the possibility to upload all file formats
- In addition, there should also be the possibility to make handwritten notes (sketch), which we solved with a canvas

Since my tasks were very different, I worked in all packages. Where I have made the following changes in the following packages:

- **model**:
  - package /**attachment**: Here the attachments were implemented based on the factory pattern. For this there is an AttachmentFactory, where we distinguish 2 in the context of this project: File and Sketch
  - Task and Checklist classes to handle the deletion of subtasks
- **view**:
  - package /**adapter/recyclerviewadapters**: to handle the deletion of the subtasks I also edited the two adapter classes checklisttask and checklisttaskgroup
  - Regarding attaching files and sketch I have edited mostly the classes AddEditTaskActivity, AddSketchActivity, Canvas and FileUtils
- **viewmodel**:
  - In this package I made some changes, where I then had to undo many of them, because we then

    1) decided to use a different pattern (for deleting subtasks: observer pattern instead of strategy pattern)

    2) deleting children did not work as I had planned (with Intent)

  - I have edited the DataBaseControl class for deleting the subtasks
- **layout**:
  - I also extended the layout activity_add_task with two buttons for attaching files and sketches and added a new layout activity_add_sketch for creating sketches

Member 3:

The responsibility of member 3 was to implement notifications functionalities.
Our approach was to implement Strategy design pattern in order to create different types of the notifications. For the appointment upcoming notification, we decided to combine Observer and Strategy pattern. Since new notification needs to be created when date of the appointment task is close to today's date. In our case, when list of appointment tasks is updated, our observer will be notified and invoke the action to respond to the notification.

User selecting notification preferences and triggering notifications.

To save the user's notification preferences we used shared preferences. In the MainActivity class, the **getNotificationPreferences()** method is used to retrieve the

user's notification preferences that were previously saved using the **saveNotificationPreferences()** method. if the preferences are not saved yet, it will return true for all the preferences, meaning that the user will receive all notifications by default. **saveNotificationPreferences()** method writes the new values to the SharedPreferences file and **getNotificationPreferences()** method retrieves the values from that file.
notification_preferences.xml. In this layout file, we added checkboxes for each notification type (popup, email) and action (create, update, delete), and a button for the user to save their preferences.
showNotificationPreferencesDialog() that inflates the notification_preferences.xml layout and display it to the user.

Whenever new settings are saved, updated notification preferences are also stored in CenterControl class. In the CenterControl, whenever an action is triggered, we check if notification preference is enabled for that action, if yes, we additionally check which type of notification is enabled and based on that we create a new notification. Further about Notification Strategy implementation can be found in section 1.3.8.

```
public void deleteTask(AppointmentTask appointmentTask){
    this.dataBaseControl.delete(appointmentTask);
    if(notificationPreferences.get("isDeleteNotification")) {
        if (notificationPreferences.get("isPopupNotification")) {
            notificationSender.setNotification(new PopUpNotificationStrategy(ui.getContext(),
                                               message: "Appointment task is successfully deleted!"));
            notificationSender.sendChosenNotification();
        }
        if(notificationPreferences.get("isEmailNotification")) {
            notificationSender.setNotification(new EmailNotificationStrategy( emailAddress: "team206@gmail.com",
                                               ui.getContext(),  message: "Email Notification has been sent."));
            notificationSender.sendChosenNotification();
        }
    }
}
```

Figure 1.2: Example how based on notification preferences, correct notification is sent.

Appointment Upcoming Notification

For implementing notification for upcoming appointment we chose to use alert notification. We created AppointmentObserver and there implemented overriden update() method that checks each appointment task in the list, comparing the due date of each task with today's date, and check if the deadline is two days from now or today, it triggers the pop-up notification reminding about the upcoming task.

Since in ListViewSetupHandler, new AppointmentTaskAdapter class is already being created, we decided this is a good place to add Obesrver, in this case AppointmentObserver to AppointmentTaskAdapter. When an updated tasks list is set in setAllTasks() method in AppointmentTaskAdapter class, our observer will be notified and the list of the tasks will be passed to its update() method. It will be

notified whenever a list view is opened and whenever new appointment task is added. Details about Observer pattern in general can be found in section 1.3.5.

Member 4:
the keyword here was refinement with iteration, I first tried to plan and design solutions for whole app then through research and then implementation I have tried to refine the design with what is most logical for an android app.
In order to guarantee low coupling between view components and the viewmodel/model, I had decided to only use databaseControl instance in view elements which handles all logic and connections behind the viewmodel and stored data in db and model

For example I had to incorporate Template design pattern to the fragments that were holding both views, and I did this with coming up with the idea of creating an abstract class ViewSetupHandler with template method loadView() which holds steps up setting up a fragment with the correct layout, db instance, observers for updating data, and my implemented/expanded adapters for each task/view with to allow interactivity with views through the app.
in addition I had to expand other parts such as CustomObservers and Table datas/daos etc., with the same design pattern used by others to implement observers so hiding a task through list view is possible with persistence (meaning the state of the tasks is stored and recovered on next startup).

For the decorator design pattern in my refined designs in order to achieve separation of concerns I decided to create two decorators extending each SetupViewHandler which are the concrete classes of the template design pattern. and for each decorator I created concrete ThemeDecorators to both achieve decorating (adding extra functionalities to object without changing the original one) with a new darker theme.

Member 5:

The responsibility for Member 5 was to import and export a tasklist. For the Export of all the tasks, we chose XML and JSON as file formats. The exported files get stored in the downloads folder of your device. For the import of tasks, a File Picker gets opened, where you can choose either a XML or JSON file. In the spinner on the main screen of the app, one can choose the file format (either JSON or XML) for importing and exporting.

Due to the reason, that I already spent much time designing the architecture and choosing the right libraries, I did not do that much changes with regard to the architecture. But still, I needed to change the chosen technology stack, because I did not try the chosen libraries practically in my code during DESIGN. For more details on the changes I did, please have a look at section 1.2.

For details on the Adapter and Factory pattern, please have a look at section 1.3.

For the decision to compare the imported task either with the id or the name, I chose the id. With this approach, the name can still be changed. But there can be possible duplicates after importing, when the id has changed, but everything else from the task is the same.

1.1.1 Class Diagrams

Member 1:

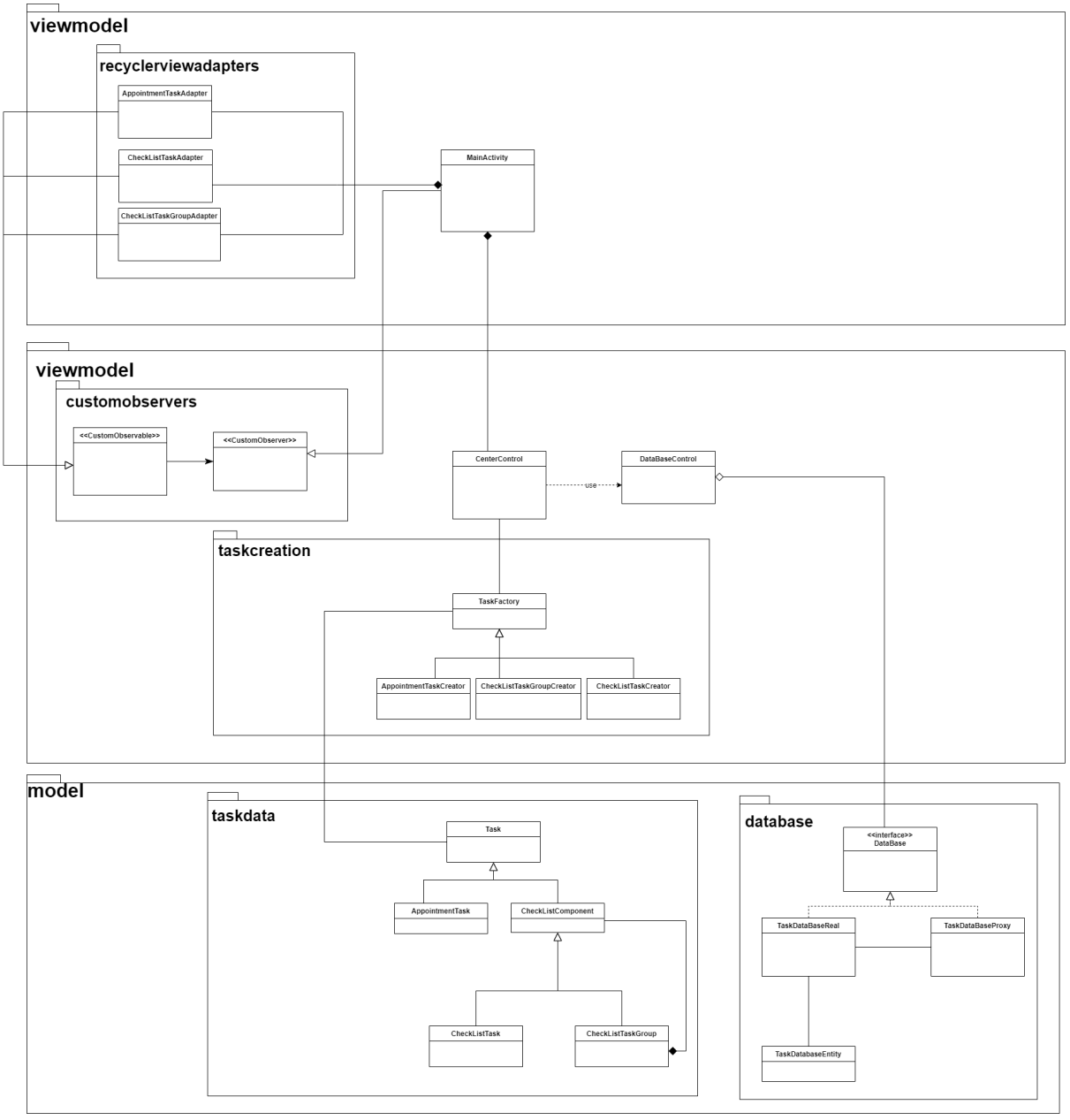


Figure 1.4: My responsibility before Merging and modified by others

All the responsibility under Member 1 was done 3 days after new year. The entire code was supported with unit tests, espresso ui test(including dbs here) as well as exception handlings, however after structure changes with calendar view, all my

Espresso UI tests are useless (see ***section 2.6***). The idea is to store all 3 different tasks into the Room DBS using 3 different table, thus 3 different adapters.

- To create different task, a Factory will handle it.
- To display in the UI, RecyclerView was used, and 3 different adapters are concated (also provided by RecyclerView) to feed in the RecyclerView. All tasks are being wrapped into LiveData, which means whenever changes happened, LiveData will automatically update the changes. There is a composite inside of taskdata package if you look closly, it is made for supporting subtasks (i.e. CheckListGroupTask could have multiple CheckListTask as children). I have to mention the way I store childrens in the dbs, I think it is worth mentioning. The way I did was to convert the subTaskList into a Json String and store it in one column in the CheckListGroup table, this means updating the children means first removing the child from the subTaskList, then insert the updated child in, lastly update the whole subTaskList.
- For updating/deleting/inserting is all being handled by DataBaseControl via CenterControl. CenterConrol is the first layer that communicates with the UI component, DataBase is the second layer that act as an interface for all the operations, that hides all the logic. TaskDataBaseProxy is the third layer that acts as a remote proxy to limit functionaily as well as a indirection layer.
- For updating the common properties, there will be a custom query that sits in every table, client will only need to call updateAllStatus from CenterControl, the rest is being handled in sub layers.

  Member 4:

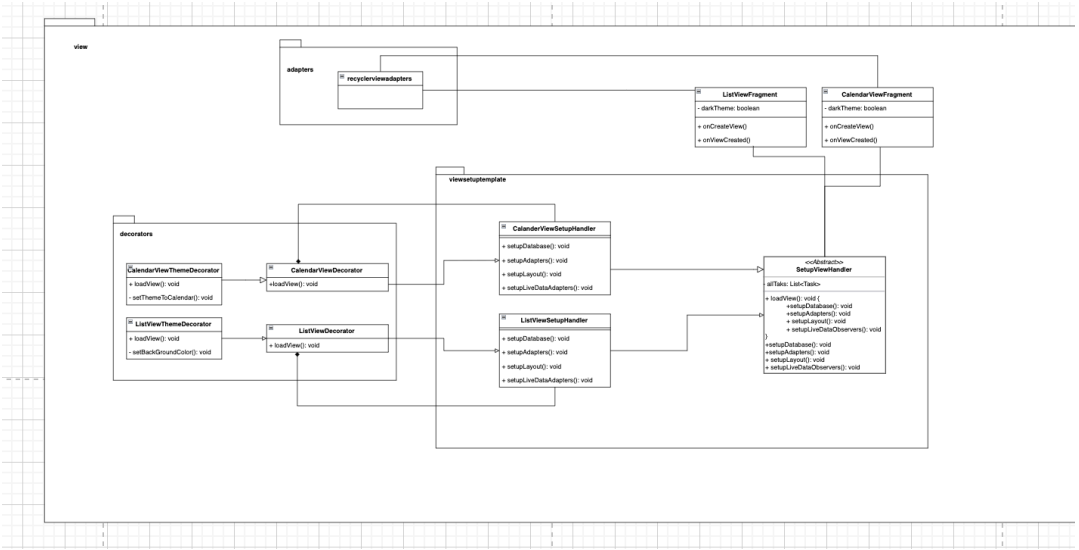


Figure 1.5: My responsibility before Merging and modified by others

the decorator and template method

1.1.2 Technology Stack

Android Studio: Android Studio Dolphin | 2021.3.1 Patch 1 (VM: OpenJDK 64-Bit)

{Android Room: version 2.4.3, Gson: 2.8.5}

- persistant storing objects in dbs
- exporting and importing json files to our application
- Room dbs youtube tutorial:
- https://www.youtube.com/watch?v=ARpn-1FPNE4&list=PLrnPJCHvNZuDihTpkRs6SpZhqgBqPU118&index=3 (full playlist of how to deal with Room Database)

{RecyclerView: 1.2.1, CardView: version 1.0.0} - used for displaying tasks on the UI

{Espresso: 3.5.0} - UI testing

For **XML** exporting and importing we used the following libraries:

- https://developer.android.com/reference/org/xmlpull/v1/XmlSerializer
- https://developer.android.com/reference/org/xmlpull/v1/XmlPullParser

We used the mentioned libraries for XML importing/exporting, because they are already included in android, so long term support should be provided and this way, we are not dependent on external libraries.

For importing of **json** to a list of tasks, we added the following class as a newly created Java class to our project:

https://github.com/google/gson/blob/master/extras/src/main/java/com/google/gson/typeadapters/RuntimeTypeAdapterFactory.java

With this class, we are able to deserialize subclasses of the task and the checklistcomponent classes with **Gson**.

sources for calendar view:
https://github.com/codeWithCal/CalendarTutorialAndroidStudio/tree/master/app

1.2 Major Changes Compared to DESIGN

Member 1:

- **Deletion/Updating of parent and/or subtasks:**

During DESIGN phase, we planned to use strategy pattern for handing update and deletion of the parent and its subtasks. However during implementation we discovered that it does not really have the structure of a strategy, and it seems very forced and unnatural to have them, since we are using Room DataBase. Room DataBase handles the deletion and update (as well as inerting) Tasks only by “executing” queries with Dao interfaces. A strategy pattern here seems overkill, and added more complexity, therefore we decided to completely abandon it. An Observer pattern will be used instead (will be discussed in ***section 2.1.4-2***)

- **Proxy Pattern:**

During DESIGN phase, the Proxy Pattern was mainly consider to be a cache proxy, where the all the task in cached in the proxy, and all insert/update/delete will be perform on the proxy instead of the actual database. However we found out that the LiveData working with Room Database almost handles everything (including executing inert/update/delete in the background using AsyncTask), so there is no need for the cache proxy anymore. The Proxy Pattern could still exist at the same place, however, it now needs to act like a remote proxy. It now provides an additional indirection layer to the real dbs, “client” now could only communicate via proxy instead of the real dbs.

Member 2

- **Factory Pattern**: During the DESIGN phase we have designed the handling of the attachments with a simple interface. While implementing it, we realized that a factory pattern is well suited here due to the fact that it is a creation pattern and we have different types of attachments:

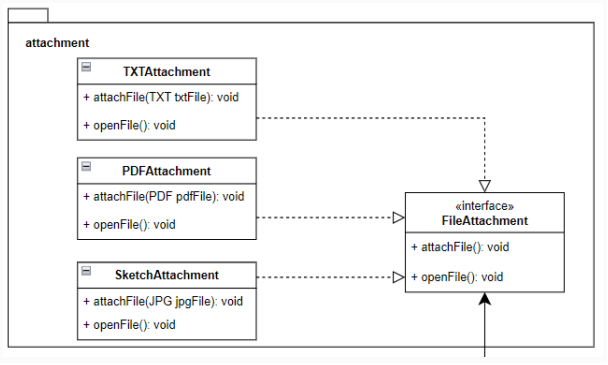


Figure 1.6: Factory Pattern for Attachments

- In the process of the implementation phase we even found out that we can merge the two file types pdf and txt under the name file. Due to the similarity of the attachment types the factory was not as extensive as planned but could still be utilised here.

Member 3:

There were no many major changes in member 3 responsibility.
Strategy pattern: CenterControl was planned to act as a Context in this pattern, however, we decided that it makes more sense to introduce another class that would act as a context and be used just for the notifications functionality as CenterControl would be doing many things. New class that was created is NotificationSender.
Notification preferences were planned to be implemented in Settings, but at the end they were implemented in the MainActivity class.
Observer pattern: Regarding the implementation of upcoming appointment notification, observer pattern was implemented a bit different than it was initially planned. I introduced new concrete observer called AppointmentObserver to manage tasks related only to the appointment notifications.

Member 4:

Multiple changes had to be done to Our first version of UML, here I will explain the ones under my responsibility:

-**template method design pattern:**
in the original version ListView and CalendarView which extended MainView supposed to hold the views themselves. but Since I wanted 2 tabs two hold each view I had to create two fragment classes which live on main activity. and for implementation of template design pattern I renamed Mainview to (abstract) ViewSetupHandler which defines abstract functions of the template method and the template function loadView(), then two different concrete classes extend it and implement each function based on their use case, this classes are CalendarViewSetupHandler and ListViewSetupHandler. which are used in their relative fragment.

-decorator design pattern:
as for the decorator pattern, I have added two abstract decorator classes for each ViewSetupHandler that based on theme chosen by the user decorates the object with a function to change background of views dynamically (in the concrete CalenderViewThemeDecorator and ListViewThemeDecorator)

other changes that I had to make was to the database with new tables/columns with respect to its related design pattern classes done by other members (observer/proxy etc which I had to expand with correct pattern) so the users choice on theme and hiding/unhiding tasks are persisted

Member 5:

In terms of Class Diagram no changes were necessary in comparison to DESIGN, due to excessive planning in the architecture creation. For the first time, I experienced the benefits of planning in advance before implementation, because implementation felt faster, but technical errors during implementation of the import especially slowed the process significantly.

Only in implementing the functionality of Import/Export of XML/JSON I experienced many difficulties in terms of choosing the right library or doing it manually. In the end, what I changed was that for XML, instead of JAXB, I used the Android libraries for XML import and export (for details, please have a look at the Technology Stack section of this report). JAXB is not suitable for Android applications, or requires much effort to find a workaround to use JAXB in Android environments. I tried to find alternatives for JAXB for a long time, until I found the Android XmlPullParser and XmlSerializer, which perfectly suited my use case. Although these libraries, especially the XmlPullParser are not that easy to use and require a lot of time and debugging to make them work, I managed to make the import and export of XML run. For the Json part, I used GSON, which is quite easy to use for exporting. Nevertheless the importing of inherited classes and its subclasses was quite a challenge. For this, I used a class from the official Github of GSON to register my subclasses and the deserializer of Gson can recognise my subclasses. Also as a prerequisite, I had to add a type parameter to the Task class, to differentiate between the different subclasses, as stated in the following website (https://www.baeldung.com/gson-list).

1.3 Design Patterns

1.3.1 Factory Pattern (Member 1, 2)

1) The Factory pattern solves the problem of creating different type of tasks. The implementation is not followed exactly from the lecture, some inspirations were taken from Tutorialspoint (https://www.tutorialspoint.com/design_pattern/factory_pattern.htm). However the fundamental concept is the same, which is to let subclasses decide which type of task to instantiate.

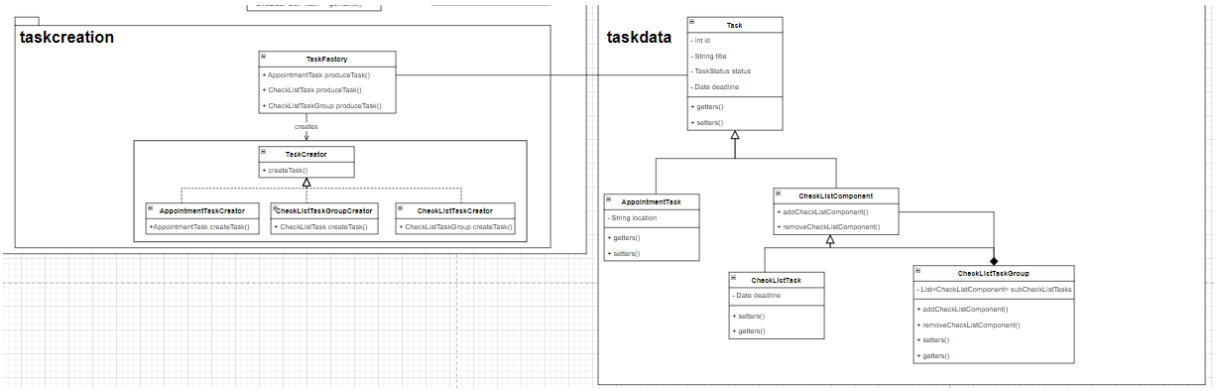


Figure 1.3.1: Factory pattern and Composite pattern UML

Figure 1.3.1 above shows exactly how the pattern was implemented. The ***taskcreation*** package contains the "Factory" side, the ***taskdata*** package contains the "Product" side. The concrete factories described in the lecture, in our case is the AppointmentTaskCreator, CheckListTaskCreator and CheckListTaskGroupCreator, and the Task Factory uses all those creators to instantiate concrete tasks (Appointment, CheckList, CheckListGroup task on the right).

Code snippets:

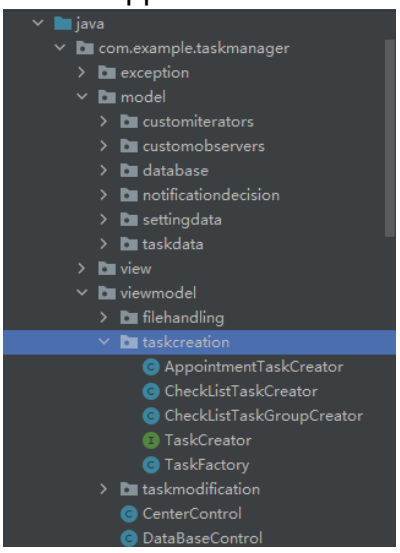


Figure 1.3.2: taskcreation package

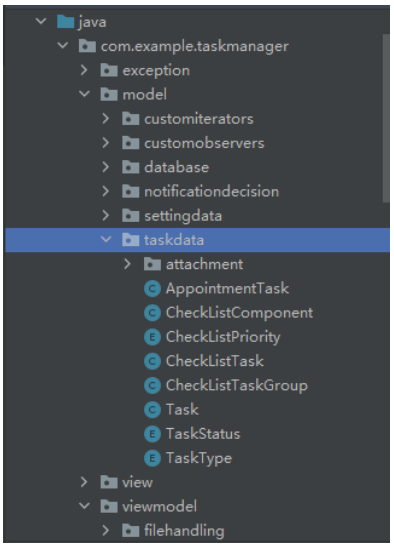


Figure1.3.3: taskdata package

2) Within Member 2's responsibility, Factory Pattern was used to handle attachments, as mentioned earlier.

1.3.2 Composite Pattern (Member 1, 2)

Composite Pattern solves the problem of the a task can be composed of subtasks. It solves this hierarchical structure presented by the Task and it sub-tasks. To be more specific, CheckListTask act as a leaf, CheckListGroupTask act as the composite and CheckListComponent act as component, which means the Group could contains subtasks(which could be another group, or a CheckList), but not the CheckList (leaf). As Figure 1.3.1 shows above, the ***taskdata*** package also contains the composite pattern, and is closely connected to the Factory pattern.

1.3.3 Facade Pattern (Member 1)

Facade Pattern is the most straight forward pattern out of all. It exist in many places, however the most obvious one is the CenterControl, where it handles insert/update/delete as well as updating all task's common property (see **figure 1.4**). It is the first "layer" the client interacts.

1.3.4 Proxy Pattern (Member 1)

More specifically, it is the Remote Proxy Pattern we are using, that mainly provides an additional indirection/layer and reduces complexity for client to use (since client does not have to know how the internal structure of the real object works). It also could create extra security, to prevent some critical functionality being used from the client. In our case, we wrapped the real database (Room) with a proxy (see in Figure 1.3.4).

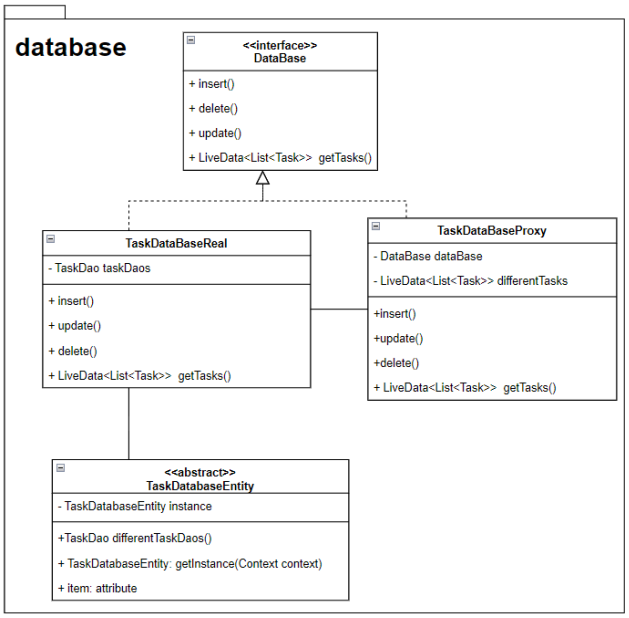


Figure 1.3.4: Proxy Pattern UML

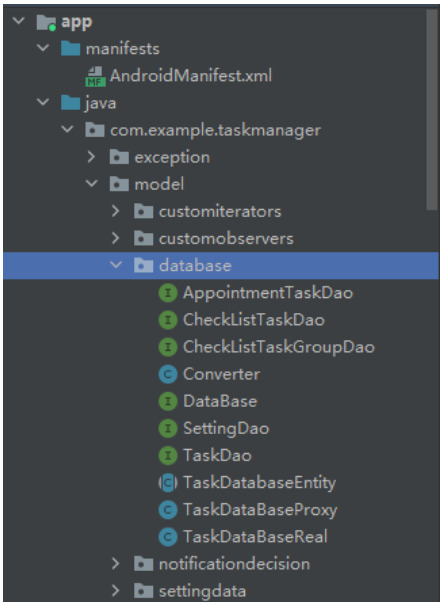


Figure 1.3.5: database package contains all Room database (including proxy pattern) related classes

The client will access the TaskDatabaseEntity via DataBase. In our case, client can only use the TaskDataBaseProxy to insert/update/delete a task in the dbs. In more detail, the proxy also does not have to know how the insert/update/delete are being

handled (i.e. using AsyncTask), since those are the responsibility of the real subject (TaskDataBaseReal). Further more, in the future, there might be more functions implemented in the database, the proxy could limit the access of a certain functions for certain client.

Code snippets:

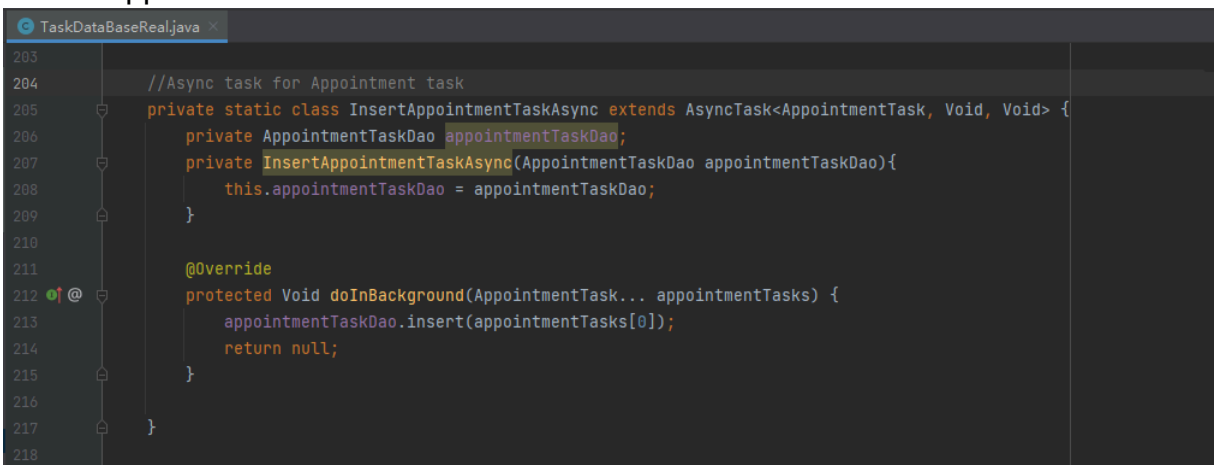


Figure 1.3.6: real subject needs the AsyncTask in order to insert a Appointment in the background

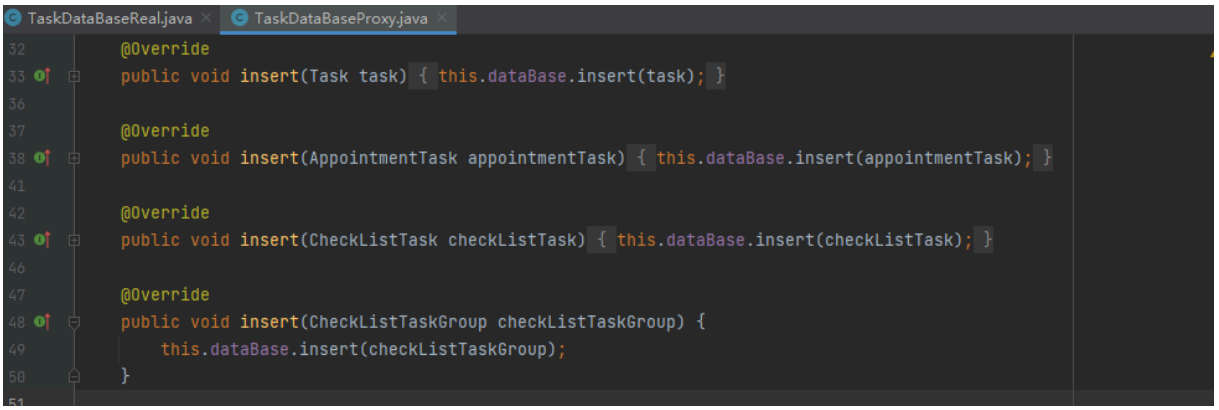


Figure 1.3.7: proxy does not need to know how the insert happened (does not need to know AsyncTask)

1.3.5 Observer Pattern (Member 1, 2, 3, 4)

There were 3 place where the Observer Pattern being used by different members.

1. **Upcoming appointment notification**
   The observer pattern was used to notify the AppointmentObserver class when an appointment task list is created or updated, but also when the application starts and the appointment list is set so it can check if there's an upcoming appointment and send a notification if necessary.

The observer pattern consists of four main components:

- The subject: The subject is the object that is being observed, in our example, the **CustomObservable** class is the subject.
- The concrete subject: The concrete subject is a concrete implementation of the subject. In our example, the **AppointmentTaskAdapter** class is the

concrete subject. it stores a list of observers and notifies them when its state changes by calling the notifyObservers() method.
- The observer: The observer is an object that wants to be notified of changes to the subject's state. In our example, the **CustomObserver** class is the observer.
- The concrete observer: The concrete observer is a concrete implementation of the observer. In our example, the **AppointmentObserver** class is the concrete observer. It observes the AppointmentTaskAdapter objects and checks if there's an upcoming appointment to send a notification.

When the list of tasks in the AppointmentTaskAdapter is created or updated, it calls the notifyObservers() method and passes list of all appointment tasks as an argument. The AppointmentObserver then updates its state by calling the update() method of the Observer interface. Inside the update() method, it checks if there's an upcoming appointment, and if there is, it sends a notification.

The observer pattern allows multiple observer objects to be notified of changes to the subject's state, without tightly coupling the subject and observer classes. The main benefit of using the observer pattern is that it allows you to define one-to-many dependencies between objects, so that when one object changes state, all its dependents are notified and updated automatically.

2. **For passing the data of a task to AddEditActivity (Member 1, 4)**

First you may ask, why is it add and edit? Because the way of adding a task and updating it is extremely similar. Adding a task requires an different activity to open, so that user could input the values in. Editing a task will requires the exact same acitivty, however it needs the data from the task which user clicked to populate onto those “input” fields.

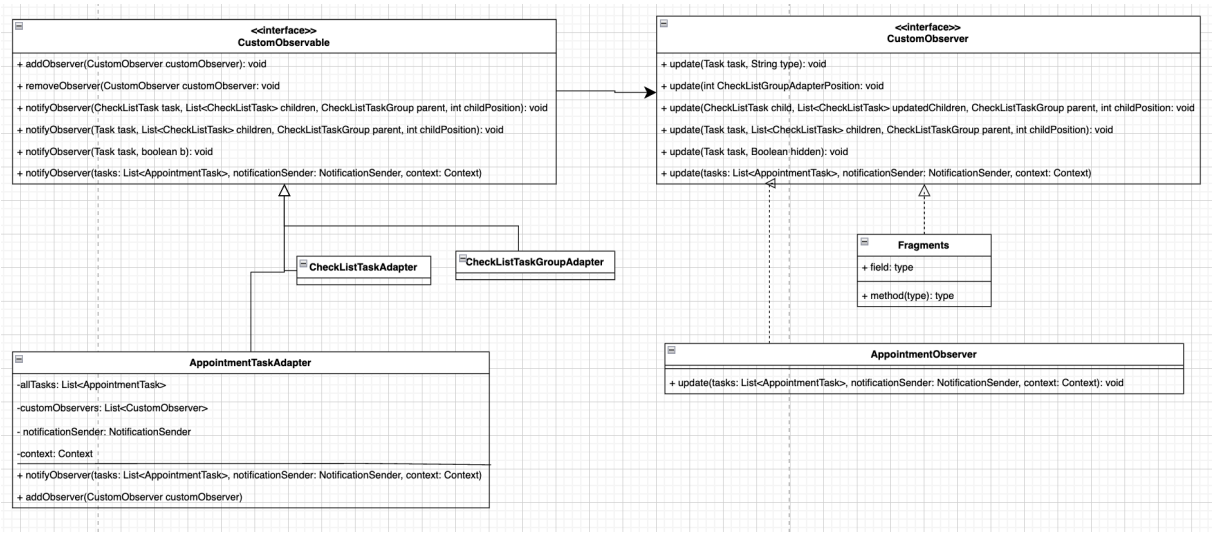


Figure 1.3.8: Highlevel view of the Observer Pattern

So instead of Strategy pattern from the DESIGN phase we originally planned, we used Observers and Intent passing to handle update and deleting the tasks.

The concept is to use the recyclerview adapters as Observerables to pass the correct task that user has clicked inside the adapter (since adapters know the relative position of its tasks). Then, our Fragments are Observers which will get the task, and soon as it gets it, it opens the AddEditTaskActivity and passes the necessary data from the task to populate the screen via the help of Intent.

This part was done by Member 1 and 4. In the beginning the MainActivity is the Observers and not Fragments, which means the data will be passed between MainActivity and AddEditTaskActivity (since MainAcitivity has the CenterControl, all the updating/deleting was handled with CenterControl). Then Member 4's repsonsiblity needs to have Calendar view and List view, and he used Fragments to achieve that. For that reason the Fragment becomes the Observers. and the Intent passing will be between them instead.

Code snippets:

Figure 1.3.9.a): notifying Obsersers when task is being clicked inside of one of the recyclerview adapters

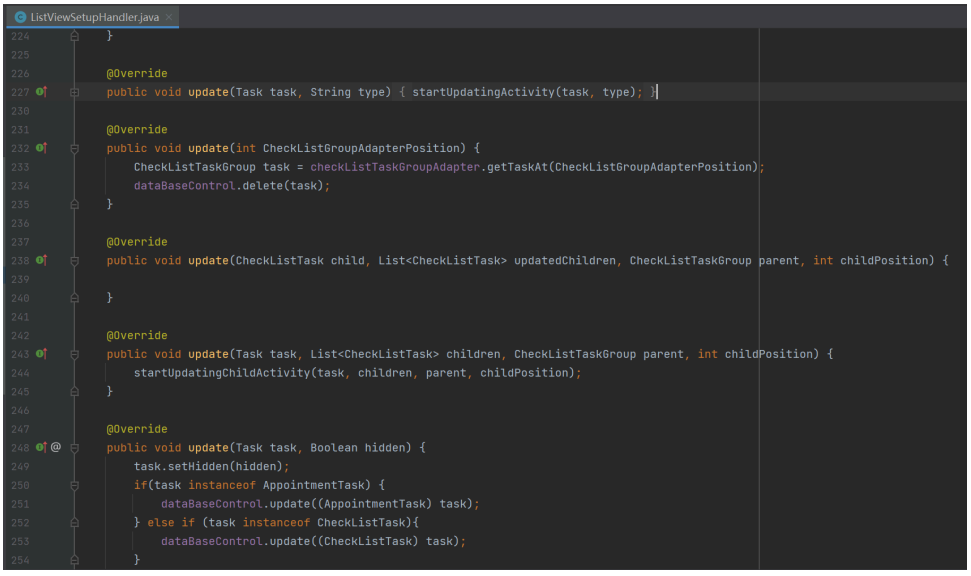


Figure 1.3.9.b): One of the fragment that act as Observers

```
public void setAllTasks(List<AppointmentTask> allTasks){
    this.allTasks = allTasks;
    notifyDataSetChanged();
    notifyObserver(allTasks, notificationSender, context);
}
```

Figure 1.3.10.a): notifying Observers when list of appointment tasks is set inside the Appointment task adapter

```
public class AppointmentObserver implements CustomObserver {
    @Override
    public void update(List<AppointmentTask> tasks, NotificationSender notificationSender, Context context) {
        for (AppointmentTask task : tasks) {
            LocalDate today = LocalDate.now();
            Date dueDate = task.getDeadline();
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(dueDate);
            LocalDate taskDueLocalDate = LocalDate.of(calendar.get(Calendar.YEAR), month: calendar.get(Calendar.MONTH)+1,calendar.get(Calendar.DAY_OF_MONTH));
            int daysUntilDue = taskDueLocalDate.getDayOfYear() - today.getDayOfYear();
            if (daysUntilDue == 1 || daysUntilDue == 2) {
                AlertDialog.Builder builder = new AlertDialog.Builder(context);
                builder.setTitle("Upcoming Appointment")
                        .setMessage("Your appointment is approaching: " + task.getTitle() + " on " + dueDate)
                        .setCancelable(false)
                        .setPositiveButton( text: "OK", new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog, int id) { dialog.cancel(); }
                        });
                AlertDialog alert = builder.create();
                alert.show();
            }
```

Figure 1.3.10.b): AppointmentObserver that acts as Observer

3. Deletion of parent and/or subtasks (Member 2)

We need observer pattern for deleting subtasks, this is done in update method in the class DataBaseControl:

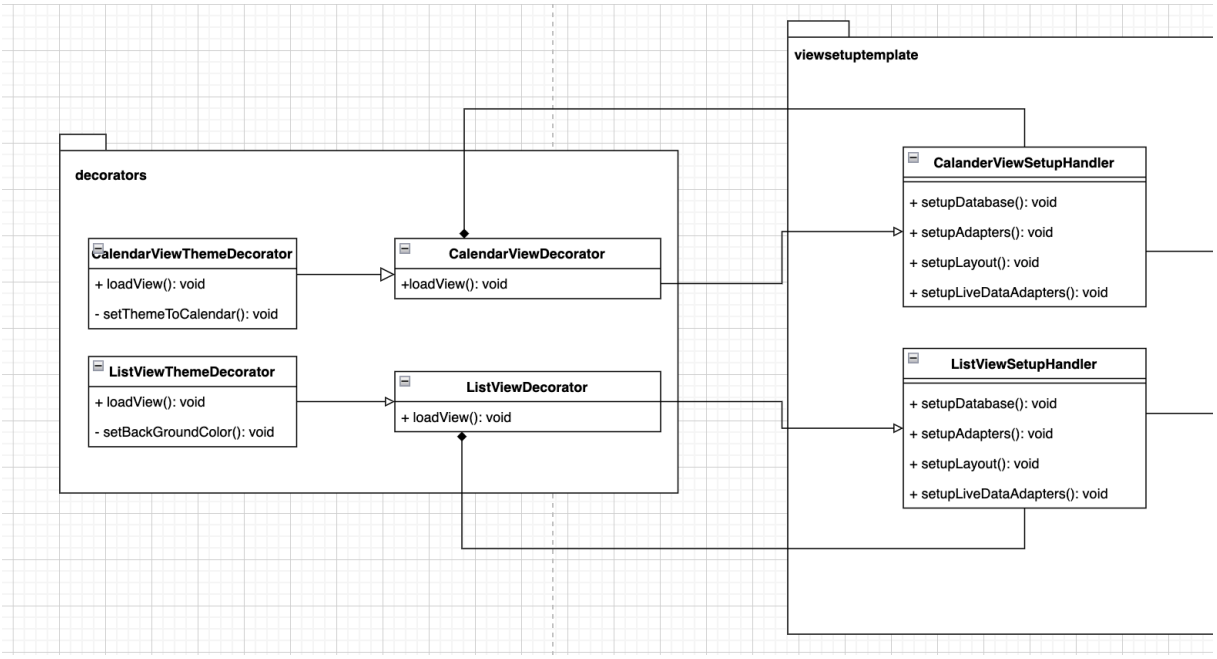
```
@Override
public void update(CheckListTask child, List<CheckListTask> updatedChildren, CheckListTaskGroup parent, int childPosition) {
    CheckListTaskGroup updatedGroupAfterDeletingChild = new CheckListTaskGroup(parent.getTitle(), parent.getStatus(), parent.getDeadline());
    updatedGroupAfterDeletingChild.setId(parent.getId());

    List<? extends CheckListComponent> newChildren = updatedChildren;
    updatedGroupAfterDeletingChild.setSubCheckListTasks((List<CheckListComponent>) newChildren);

    update(updatedGroupAfterDeletingChild);
}
```

Figure 1.3.11.: Deleting the subtasks

1.3.6 Decorator Pattern: (Member 4)



The decorator pattern is used so in case we need to add functionalities to an object, we don't need to make changes to the original one.
as you can see in part 2.1.6, both CalendarViewSetupHandler and ListViewSetupHandler are concrete classes that extend SetupViewHandler, and we want to decorate their loadView() with a function that changes the background color of the fragments based on users persisted settings. We achieve this by creating a class that extends the handler for example CalendarViewDecorator should extend the CalendarViewSetupHandler and

```
56              CalendarViewThemeDecorator calendarViewThemeDecorator =
57                      new CalendarViewThemeDecorator( calendarViewFragment: this, view,
58                      calenderViewSetupHandler);
59
60              calendarViewThemeDecorator.loadView();
```

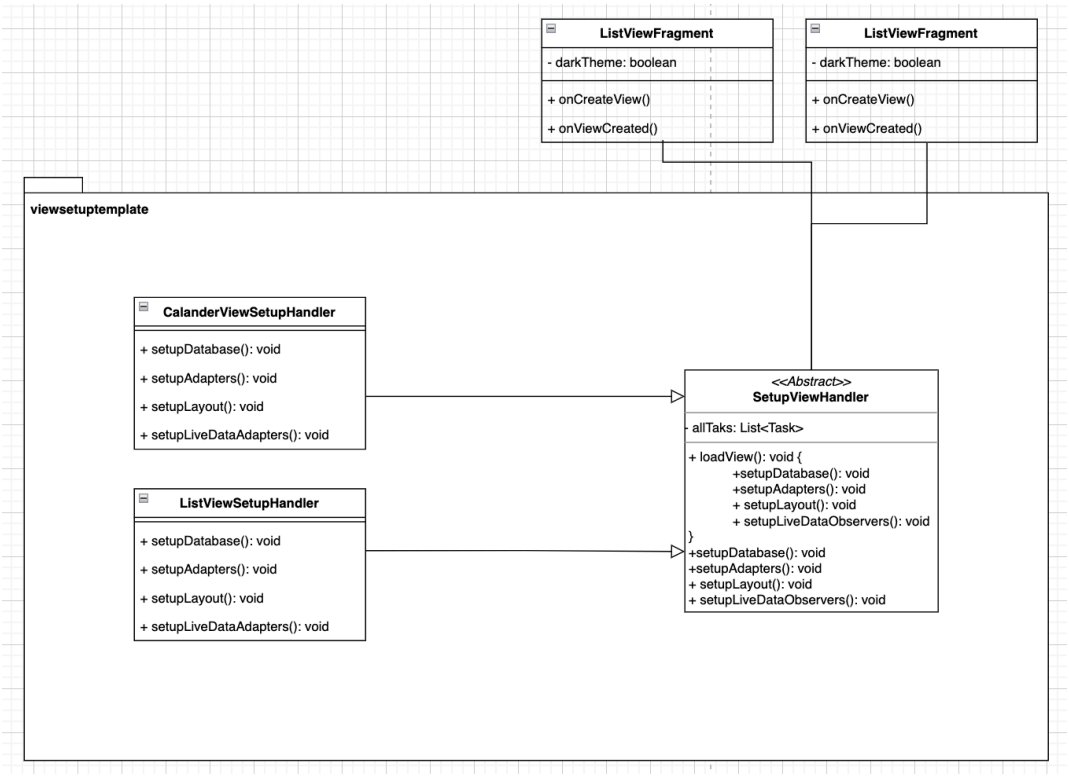
```
8   public class CalendarViewDecorator extends CalenderViewSetupHandler {
9       CalenderViewSetupHandler calenderViewSetupHandler;
10
11      public CalendarViewDecorator(CalendarViewFragment calendarViewFragment, View view,
12                                   CalenderViewSetupHandler calenderViewSetupHandler) {
13          super(calendarViewFragment, view);
14          this.calenderViewSetupHandler = calenderViewSetupHandler;
15      }
16
17      @Override
18      public void loadView() {
19          calenderViewSetupHandler.loadView();
20      }
21  }
```

```
25      @Override
26      public void loadView() {
27          calenderViewSetupHandler.loadView();
28          setThemeToCalender();
29      }
30
31      private void setThemeToCalender(){
32          LinearLayout calendarView = (LinearLayout) view.findViewById(R.id.calendar_view);
33          calendarView.setBackgroundColor(Color.DKGRAY);
34
35          TextView tv = (TextView) calendarView.findViewById(R.id.monthYearTV);
36          tv.setTextColor(Color.WHITE);
37      }
38  }
```

keep a global instance of it, then on concrete CalendarViewSetupHandler we add new function setThemeToCalendar() which will be called in it's overwritten loadView() alongside the call to original version of loadView(). (see the first figure to see the way to crate a decorated object) in the 3rd figure you can see how the original functionality of loadView alongside the setThemeToCalendar() is called in the concrete CalendarViewThemeDecorator.

The same is also done for the listView decoration, and whether or not to decorate the ViewSetupHandler is decided buy user's persisted input for settings which will be passed to the fragment by a bundle to choose.

1.3.7 Template Method Pattern: (Member 4)



I used this design pattern so that both fragments of the view can use the similar skeleton they need from using the loadView() function of Abstract SetupViewHandler, which is overwritten in the concrete classes to what each fragment needs.

```
3  public abstract class ViewSetupHandler {
4
5      abstract void setUpDataBase();
6      abstract void setUpAdapters();
7      abstract void setUpLayout();
8      abstract void setUpLiveDataObservers();
9
10     public void loadView() {
11         setUpDataBase();
12         setUpAdapters();
13         setUpLayout();
14         setUpLiveDataObservers();
15     }
16 }
```

```
32     @Override
33     public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
34         super.onViewCreated(view, savedInstanceState);
35         Bundle args = getArguments();
36         assert args != null;
37         darkTheme = args.getBoolean( key: "darkTheme");
38
39         ListViewSetupHandler listViewSetupHandler;
40
41         if (darkTheme){
42             listViewSetupHandler = new ListViewThemeDecorator( listViewFragment: this, view,
43                     new ListViewSetupHandler( listViewFragment: this, view));
44
45         }else {
46             listViewSetupHandler = new ListViewSetupHandler( listViewFragment: this, view);
47         }
48
49         listViewSetupHandler.loadView();
50     }
```

as you can see on the right figure, the ListViewFragment when created, instantiates a new ListViewSetupHandler and at line 49 calls the template loadView() which is customised to needs of this view inside concrete ListViewSetupHandler, note that the same happens inside The CalendarViewFragment with its concrete class

CalendarViewSetup handler. also the bundle is used to get whether user has chosen dark/light theme which is passed from mainActivity and to choose whether decorator is needed or not

1.3.8 Strategy Pattern (Member 3)

Our strategy pattern consists of three components:

- The strategy interface, in our case **NotificationStrategy** interface, which defines the common interface for all concrete strategies.
- The concrete strategies, in our case **PopUpNotificationStrategy** and **EmailNotificationStrategy** classes, which provide specific implementations of the strategy interface.
- The context, in our case **NotificationSender** class, which is responsible for choosing the appropriate concrete strategy to use and delegating the work to it.

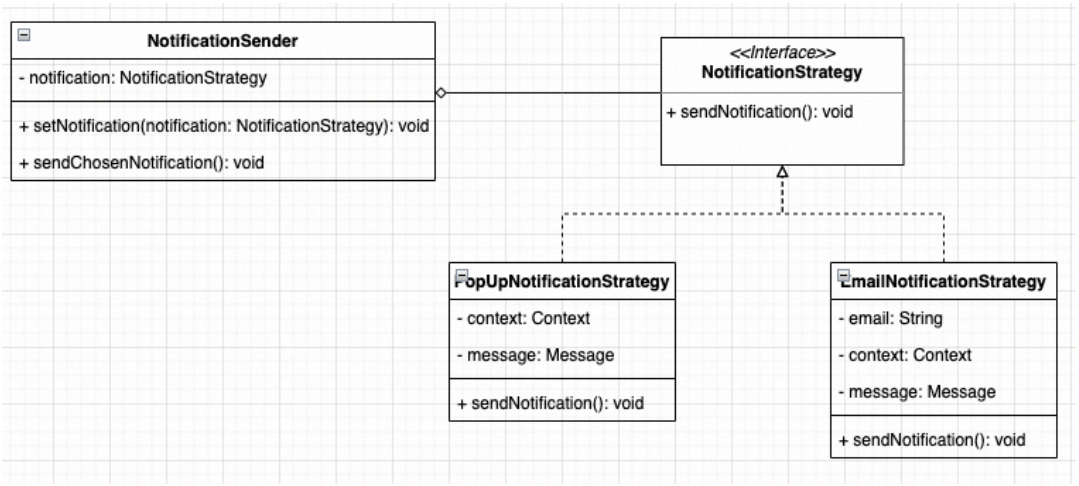


Figure 1.3.16: Strategy Pattern UML

The NotificationStrategy interface represents the "strategy" component. It defines a common interface for all concrete strategies, in this case it defines the sendNotification() method that all concrete notification classes must implement.

```
public interface NotificationStrategy {
    public void sendNotification();
}
```

Figure 1.3.17:NotificationStrategy class

The concrete strategies, PopUpNotificationStrategy and EmailNotificationStrategy classes, are implementations of the NotificationStrategy interface. They provide specific implementations of the sendNotification() method to display a pop-up notification or send an email notification respectively.

```java
public class EmailNotificationStrategy implements NotificationStrategy{
    private String emailAddress;
    private Context context;
    private String message;

    public EmailNotificationStrategy(String emailAddress, Context context, String message) {
        this.emailAddress = emailAddress;
        this.context = context;
        this.message = message;
    }
    @Override
    public void sendNotification() {
        Log.d( tag: "EmailNotification",  msg: "Email sent to " + emailAddress);
        //Toast message. check alert
        Toast.makeText(context, message, Toast.LENGTH_SHORT).show();
    }
}
```

```java
public class PopUpNotificationStrategy implements NotificationStrategy{
    private Context context;
    private String message;

    public PopUpNotificationStrategy(Context context, String message) {
        this.context = context;
        this.message = message;
    }

    @Override
    public void sendNotification() { Toast.makeText(context, message, Toast.LENGTH_SHORT).show(); }
}
```

Figure 1.3.18: PopUpNotificationStrategy and EmailNotificationStrategy classes

The NotificationStrategy interface and the concrete strategies are completely decoupled, which allows us to add new concrete strategies without having to change any existing code. In this way, the strategy pattern provides a way to encapsulate a family of algorithms and use them interchangeably.

NotificationSender class is responsible for deciding which type of notification to send based on the user's notification settings stored in SharedPreferences. It uses the setNotification() method to set the appropriate concrete strategy (PopUpNotificationStrategy or EmailNotificationStrategy) and then calls the sendChosenNotification() method to send the notification.

```java
public class NotificationSender {
    private NotificationStrategy notification;

    public void setNotification(NotificationStrategy notification) {
        this.notification = notification;
    }

    public void sendChosenNotification() { notification.sendNotification(); }
}
```

Figure 1.3.19: NotificationSender class

1.3.9 Adapter Pattern (Member 5)

In the context of Member responsibility 5 and the import of XML or JSON files, we placed an Adapter class between the CenterControl class and the external JSON (Gson) and an Adapter between the CenterControl and the XML libraries. Both Adapter classes implement an interface called ListOfTask. This way the adapters kind of act as a redirecter, doing all the conversions of a xml or json file content to a List of tasks. In our design, we do not actually have a Adaptee class, that needs to be adapted, instead we have an external library, whose methods, we will call in the Adapter class. The CenterControl class does not even have to know which library we use for a Json importation for example.

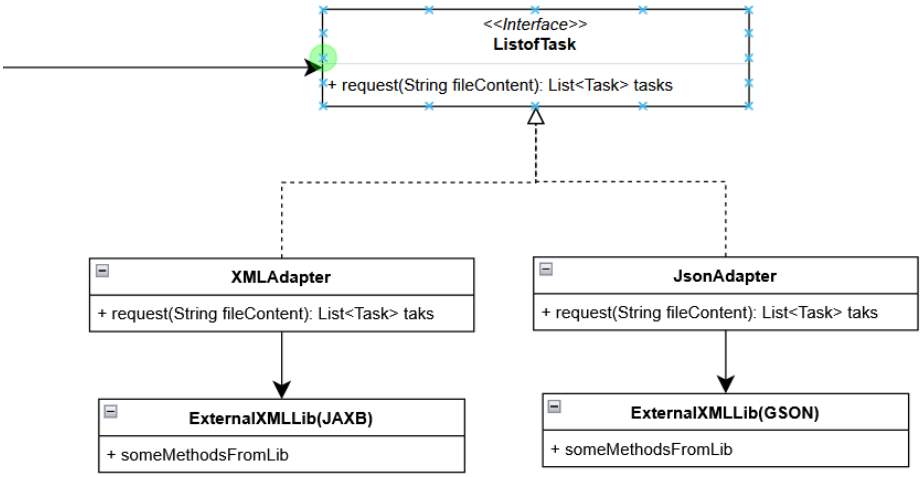


Figure 1.3.20: Adapter pattern for Import

1.3.10 Factory Pattern (Member 5)

For the exportation of all the Tasks, I used the Factory Pattern.

Depending on the passed parameter in the method produceFileCreator of the class “ExportFactory” the client (CenterControl) gets an implementation of the FileCreator interface returned.

```java
public class ExportFactory {
    /**
     *
     * @param format a string representation of the chosen export file format e.g. XML, JSON, etc.
     * @return A Class that implements the FileCreator interface.
     */
    public FileCreator produceFileCreator(String format){
        switch (format){
            case "JSON":
                return new JsonCreator();
            case "XML":
                return new XmlCreator();
        }
        return null;
    }
}
```

Figure 1.3.21: ExportFactory class (viewmodel/filehandling)

Hence the client can call the export method of the FileCreator implementation, which in our case either creates a Json file, by the **JsonCreator** class or a XML file, created by the **XmlCreator** class. This way the client(CenterControl) just calls the createFile() from the “FileCreator” interface and does not know about the concrete implementations.

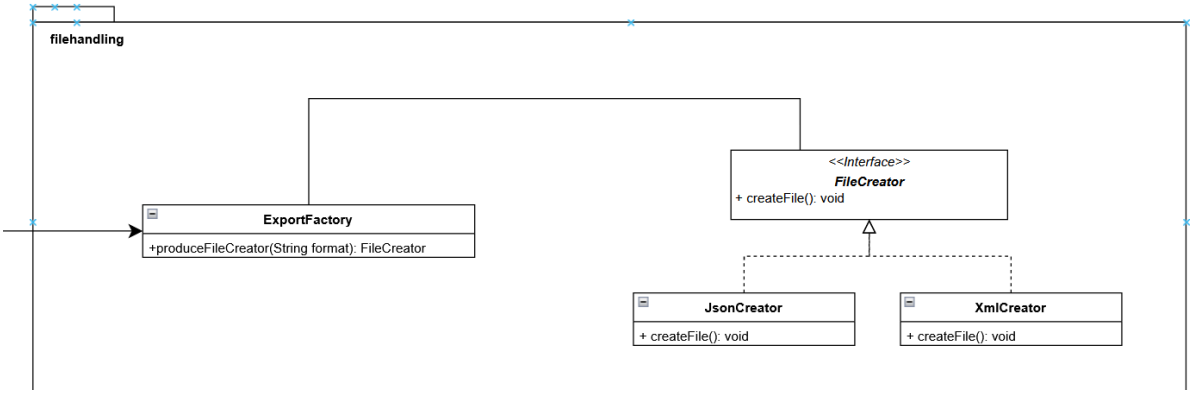


Figure 1.3.22: Factory pattern for export

2 Implementation

2.1 Quality Requirements Coverage

Since the implementation is done separately, and there are delays on members, we ran out of time to do a final code review to check on overall quality.

QR1, QR3 QR4(only on some members), QR5, QR6 (only Member 1, 4), QR7(iterator pattern was not used).

2.2 Coding Practices

Member 1:

- I took extra care on the naming convention for methods. I made sure that almost all my methods started with verb (e.g. `updateSubTasksFromParent(Intent intent), createAppointmentFromIntent(Intent intent),produceTask(String title, TaskStatus status, Date deadline, String location)`).

- I commented on methods which are less obviouly or maybe reusable for other members.
  E.g:

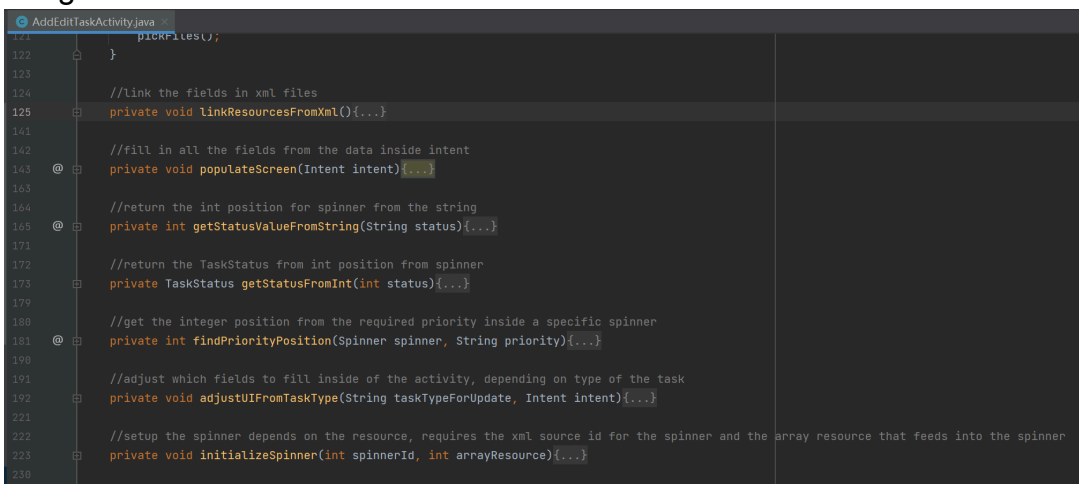


Figure 3.4.1: comments for complex methods

- To make the code more readable and less complex, I put reappearing codes into method to achieve a minum duplication of code. Also when a code comes too long and complex, I put them into a method to make it easier to understand for other team members.
  (Note: there might be duplication for Itents. the Itent class are very difficult to group, a small mistakes might causes null pointer and crashes the app.)

```
//setup the spinner depends on the resource, requires the xml source id for the spinner and the array resource that feeds into the spinner
private void initializeSpinner(Spinner spinner, int arrayResource){
    // Create an ArrayAdapter using the string array and a default spinner layout
    ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource( context: this,
            arrayResource, android.R.layout.simple_spinner_item);
    // Specify the layout to use when the list of choices appears
    adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    // Apply the adapter to the spinner
    spinner.setAdapter(adapter);
    spinner.setOnItemSelectedListener(this);
}
```

Figure 3.4.2: reusable spinner setup methods

Member 2:

- I tried to follow key naming conventions for types, variables and functions.
- The names chosen were representative of the construct they stand in for.
- Where it seemed necessary comments were made to explain the rationale behind the code.

Member 3:

- All of the most important naming conventions have been fulfilled in my code. The names chosen for variables, functions, and classes are intention revealing, meaning that their purpose is clear just from the name. They also avoid disinformation, ensuring that the names do not mislead or confuse. The names make meaningful distinctions, differentiating between similar elements in the code. Additionally, the names are pronounceable, making it easy for developers to discuss the code verbally. Finally, the names are searchable, making it simple to locate specific elements within the codebase.

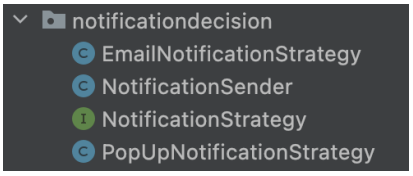


Figure 3.4.3.a): Example of naming the package and classes

- I used javadoc comments in the code, which is beneficial because it allows other developers who may read or work with the code to understand the purpose and usage of the classes, methods, and variables. Overall, including javadoc comments in the code improves the readability and maintainability of the codebase, and makes it easier for others to understand and contribute to the project.

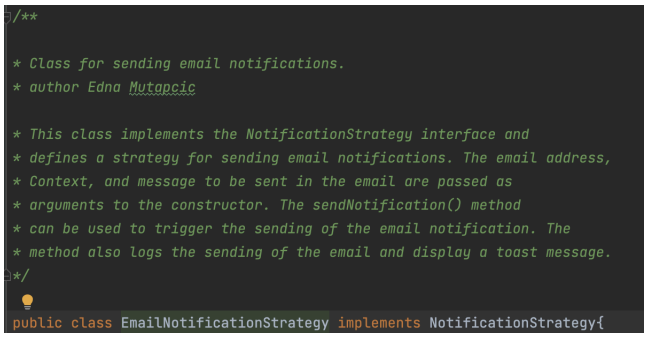


Figure 3.4.3.b): Example of the Javadoc comment

- I took care that my methods are highly cohesive operations as they are much more defect free and defects in them are less costly to fix.

Member 4:

- In order to make the code more readable for developers I have tried to name the classes/functions/variables in a way that their purpose can be understood by just knowing the name, also I tried to break up bigger and sometimes redundant code to block of smaller one purpose functions, that the bigger function calls, and the developer can read and get an idea of of what happens in each one without need of big explanation comments, I have used camel case for namings and tried to stay away from meaningless abbreviation etc.
  on top of that I have documented the classes implemented by me mostly view with 1,2 line javadoc comments so that the user gets a glimpse of the whole class.
  for formatting I have used conventional java way of formatting with consistency and meaningful packages. example of an readable code:

```
public void loadView() {
    setUpDataBase();
    setUpAdapters();
    setUpLayout();
    setUpLiveDataObservers();
}
```

Member 5:

I strictly sticked to the naming convention as stated in the lecture slides. I chose self explanatory names for methods and variables. If the measures are not sufficient, I added comments to explain my intentions. Below is an example.

```
// The old tasks get compared with the newly imported tasks via Id.
// If the id, is same the newly created task replaces the old one.
// If not, a new entry for the new task will be created
boolean foundSameTask = false;
for(Task eachNewTask: newTasks){
    for(Task eachOldTask: oldTasks){
        if(eachOldTask instanceof  AppointmentTask && eachNewTask instanceof AppointmentTask){
            if(eachOldTask.getId() == eachNewTask.getId()){
                centerControl.updateTask((AppointmentTask) eachNewTask);
                foundSameTask = true;
                break;
            }
        }
        if(eachOldTask instanceof CheckListTaskGroup && eachNewTask instanceof CheckListTaskGroup)
            if (eachOldTask.getId() == eachNewTask.getId()) {
                centerControl.updateTask((CheckListTaskGroup) eachNewTask);
                foundSameTask = true;
                break;
            }
```

Furthermore I used JavaDoc comments and formatted my code with Android Studio IDE, to make it more readable.

2.3 Key Design Principles

Member 1:
To separate the responsibility, for my implementation everything should be handled by CenterControl (see Figure 1.4). However this might change due to structural changes by other member. There are total 3

Member 2:
Separation of concerns was a key design principle I followed during the development. The handling of all file related tasks was done in the class FileUtils, which is only loosely coupled with the rest of the functionality. Furthermore sketch handling while similar was handled in the AddSketchActivity.

Member 4:
I made sure to keep separation of concerns for classes/function consistent through out the implementation, for examples creating two decorators for two views instead of using one as was initially designed, also tried to use composition over inheritance by using abstract classes, and implementing them further on concrete classes.

Member 5:
I tried to use as much abstraction as possible, by working with interfaces and super classes. I always used List<Task>, instead of List<AppointmentTask> for example. If I

need the subtype of the class, I just check with “instance of” and then simply cast to it.

2.4 Defensive Programming

Member 1:

- There are custom exceptions I made for handling errors. They are mostly unchecked exceptions, the idea is to increase the app robustness, so if theres “garbage in”, there should be no “garbage out”, the system will try to handle them as peacefully as it can to prevent the app from crashing.

```
//checks if the title, year, month, day is empty, also checks if the entered date is valid
private void inputValidation(String title, String year, String month, String day, String date){
    //trim down to see if the input is empty
    if(title.trim().isEmpty() || year.trim().isEmpty() || month.trim().isEmpty() || day.trim().isEmpty()){
        throw new EmptyInputException("input is empty");
    }

    DateFormat dateCheck = new SimpleDateFormat( pattern: "yyyy/M/dd");
    dateCheck.setLenient(false);
    try {
        dateCheck.parse(date);
    }catch (ParseException e){
        throw new InvalidDateException("invalid date");
    }
}
```

```
@Override
public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    if(item.getItemId() == R.id.save_task){
        try {
            save();
        }catch (EmptyInputException | InvalidDateException e){
            this.ui.createMessage(e.getMessage());
        }
        return true;
    }else{
        return super.onOptionsItemSelected(item);
    }
}
```

Figure 3.5.1: Here it validates if the input date is valid
(Note: inputValidation is save(), and onOptionsItemSelected will catch the exceptions and handle it locally to display a ui message)

- Intent passing requires a lof of request codes. Instead of define them all over the place, I create the IntentKeys class. Now all the keys are static final, so it could be access safely from on place when is needed.

```java
package com.example.taskmanager.view;

public class IntentKeys {
    //intend key strings to send the data to between activities when this activity is closed
    public static final String EXTRA_ID = "com.example.taskmanager.view.EXTRA_ID";
    public static final String EXTRA_TITLE = "com.example.taskmanager.view.EXTRA_TITLE";
    public static final String EXTRA_STATUS = "com.example.taskmanager.view.EXTRA_STATUS";
    public static final String EXTRA_LOCATION = "com.example.taskmanager.view.EXTRA_LOCATION";
    public static final String EXTRA_PRIORITY = "com.example.taskmanager.view.EXTRA_PRIORITY";
    public static final String EXTRA_DATE = "com.example.taskmanager.view.EXTRA_DATE";
    public static final String EXTRA_SUBTASKS = "com.example.taskmanager.view.EXTRA_SUBTASKS";
    public static final String EXTRA_TASK_TYPE = "com.example.taskmanager.view.EXTRA_TASK_TYPE";
    public static final String EXTRA_UPDATE_TASK_TYPE = "com.example.taskmanager.view.EXTRA_UPDATE_TASK_TYPE";

    public static final String EXTRA_CHILD_TITLE = "com.example.taskmanager.view.EXTRA_CHILD_TITLE";
    public static final String EXTRA_CHILD_STATUS = "com.example.taskmanager.view.EXTRA_CHILD_STATUS";
```

Figure 3.5.2: IntentKeys class

Member 2
When creating file attachments, I made sure that possible issues are recognized by checking the IDEs found by the file system and deemed openable. Should this not be the case, the attachment will not be accepted.

Member 4(Fallah):
since our goal is to implement a customer product I have used defensive measures to ensure the robustness of my implemented tasks with error handling with try catches where user input is not allowed and could create bad data or worst case crashes, for example when logic in the view for deletion of children tasks was not implemented by members, I tried to catch the exceptions thrown in the adapter with a try catch method:

```java
try {
    notifyObserver(allTasks.get(position));
}catch (Exception e){
    Log.e( tag: "nullPointerException", msg: "all edit task should happen t
}
```

another example is that My implemented ViewSetupHandlers expect the theme data (wether dark or not) to be passed as a bundle from the fragments and since after installation setting table is filled, in normal situations it is not possible that null is passed, but to ensure that I have used assertions to check whether bundle has setting data or is null so, this is not supposed to go wrong.

```java
35    Bundle args = getArguments();
36    assert args != null;
37    darkTheme = args.getBoolean( key: "darkTheme");
```

Member 5:

In general I used try catch statements in the same class. If the calling class needs knowledge about an exception, I added the throw Exception to the method signature and the calling class can catch it and subsequently handle the error, by displaying a Toast message for example. We use the principle garbage in, no garbage out, but in some cases, I returned null to indicate that something went wrong.

```java
/**
 * The sole purpose of this class is to take the json fileContent represented as a String
 * and deserialize it to a List of Tasks. This class is implementing the ListOfTask interface.
 *
 * @author Sahel Naderi
 */
public class JsonAdapter implements ListOfTask {
    /**
     * @param fileContent - String, which should contain the json
     * @return list of tasks generated from the provided String, which contains a JSON.
     */
    @Override
    public List<Task> request(String fileContent) {
        if (TextUtils.isEmpty(fileContent)) {
            return null;
        }
```

A better approach would be to add custom exceptions, but I think for our simple app, it is sufficient to return null and that is what I do in all of my code to ensure consistency. Maybe in a further patch, we will add custom exceptions.

2.5 Testing

Member 1:
For the first stage I was using **RoomDataBase.Callback** function with **logger** (can be found in TaskDataBaseEntity). The idea is to test all the Room Database functions (insert/update/delete) directly with Dao interfaces before implementing the UI component.

For the second stage I mainly used **Unit test** and **Manual Test**. I used Unit test to test some of the internal structure and correctness of my code (i.e. Factory pattern). Because some functionality (i.e DBS instantiation, insert/update/delete of a task) can not be tested with Unit test, and I am not familiar with the Espresso Automation test at that stage, to reduce time I simply used Manual Test on my physical drive along with logger to test the functionality of the UI component (i.e. adding a task, swipe to delete, display tasks on RecyclerView etc…)

For the final stage I used **Espresso test**. I used it to test the UI component without Manually input or interact with physical device. In addition I added the dbs testing with Espresso, to be more secure and professional.

Important Note: Since my UI Espresso test implementation was done before Member 4 started. The UI is being changed due to the need of CalendarView along with structure, the UI test thus does not work anymore and was all deleted in the FINAL submission. However I did spend a few days implementing it as well as a Suite to group testing them all togethere, and here are the results.

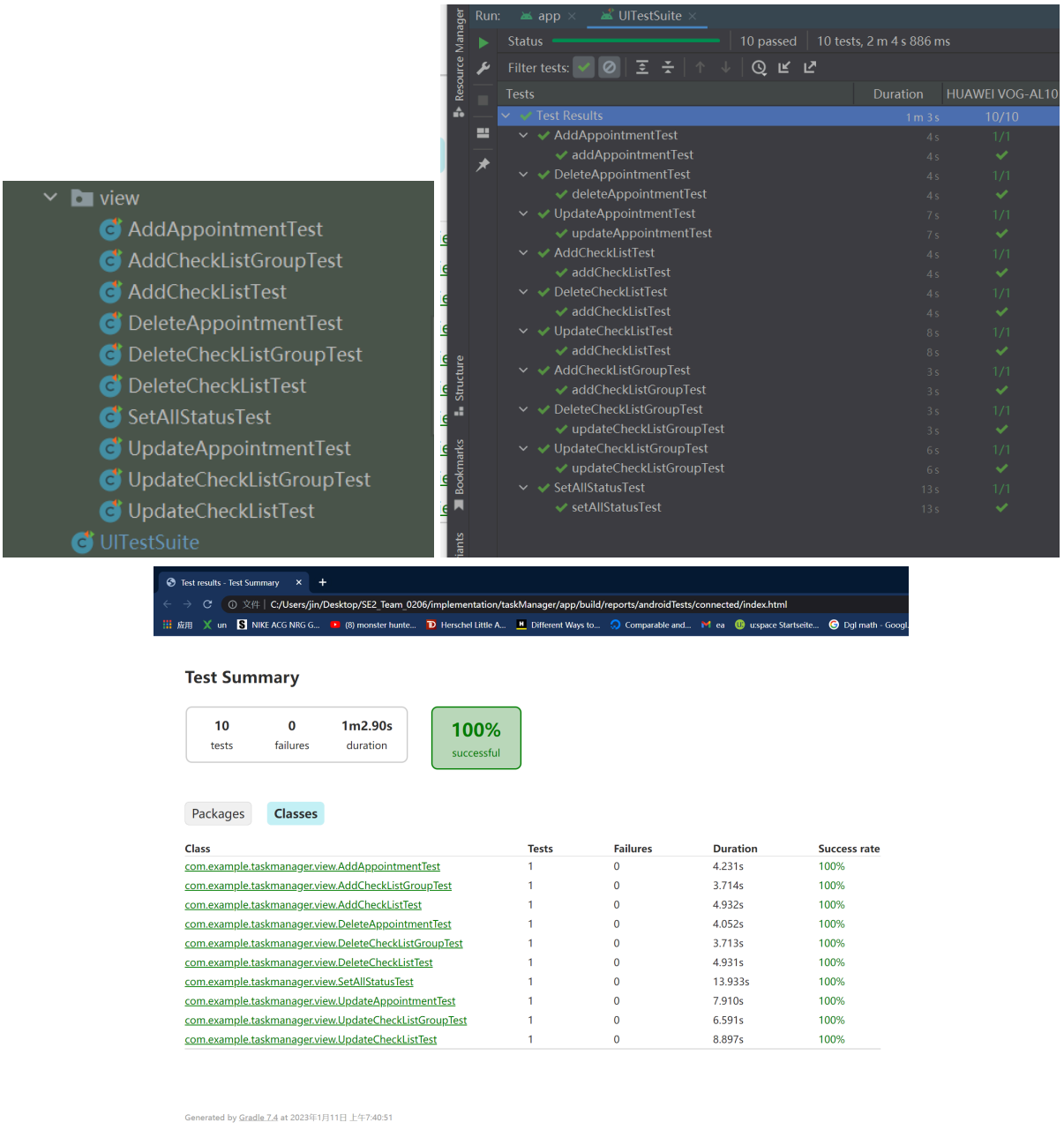
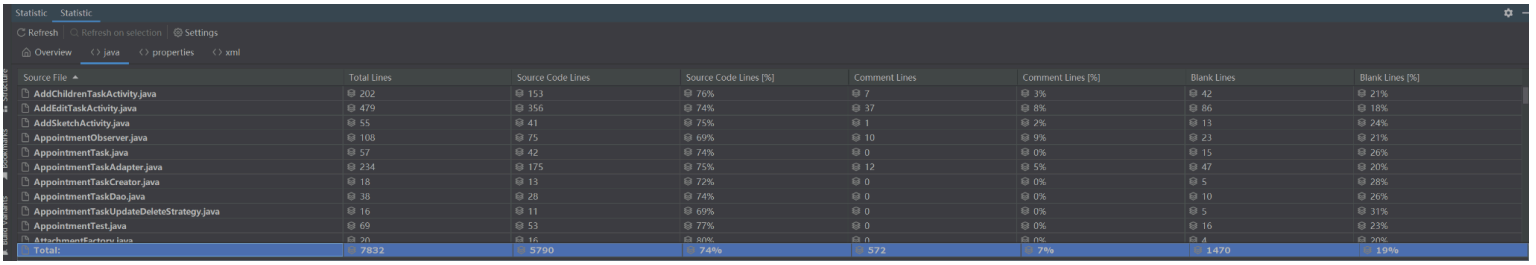


Figure 3.6.1: unit tests and espresso test

Member 4(Fallah):
in addition to regularly testing mine and other members implemented functionalities I have expanded the GeneralDBSTest class with multiple unit tests to check wether or not my implementations for persistence of data to the db correctly works and updating/getting/inserting of settings and task parameters are functionally sound,
you can see some examples below:

Member 5:

3 Code Metrics



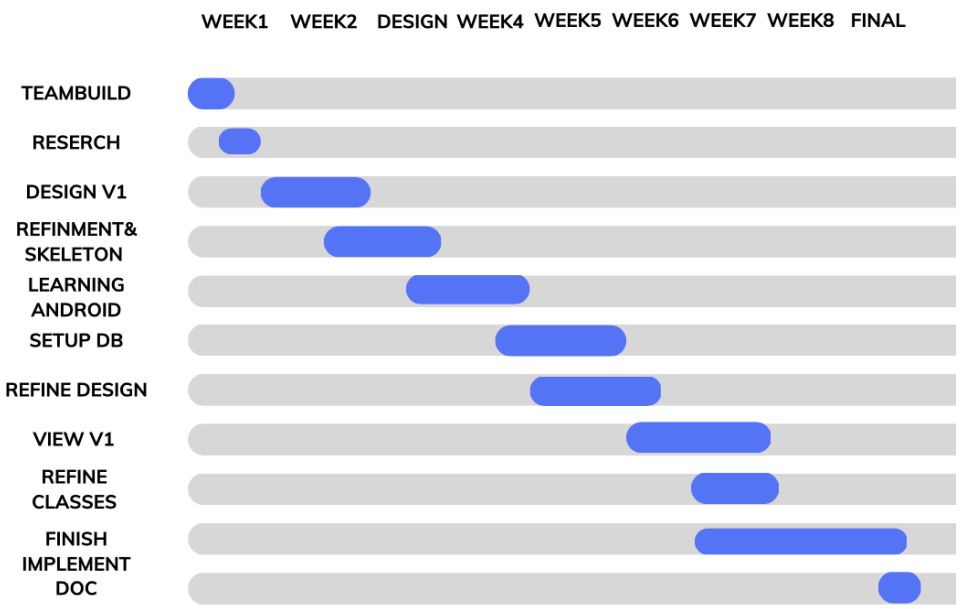
Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
AddChildrenTaskActivity.java	202	153	76%	7	3%	42	21%
AddEditTaskActivity.java	479	356	74%	37	8%	86	18%
AddSketchActivity.java	55	41	75%	1	2%	13	24%
AppointmentObserver.java	108	75	69%	10	9%	23	21%
AppointmentTask.java	57	42	74%	0	0%	15	26%
AppointmentTaskAdapter.java	234	175	75%	12	5%	47	20%
AppointmentTaskCreator.java	18	13	72%	0	0%	5	28%
AppointmentTaskDao.java	38	28	74%	0	0%	10	26%
AppointmentTaskUpdateDeleteStrategy.java	16	11	69%	0	0%	5	31%
AppointmentTest.java	69	53	77%	0	0%	16	23%
Total:	7832	5790	74%	572	7%	1470	19%

Current sate:
Total lines of code: 7832,
Commented lines: 572 (7%),
Total classes: 91
There is a signicifcant increase in classes and codes due to the complexity.

Current bugs:
hide button on tasks should not be shown in calendar view since updating the tasks happens on list view and calendar view is to filter by date

4 Team Contribution

4.1 Project Tasks and Schedule



4.2 Distribution of Work and Efforts

Contribution of member 1: (in sum)
Implementation under my responsibility: one month. (Completely finished: a week after new year)
Testing (unit test and espresso test): 3 days
Documentation: 1 day
Code review (only under my implementation): 1 day

Contribution of member 2 (Merve Sen):

tasks incl. research:

1) one type of task should be able to be composed of subtasks: 3 days
2) deletion of parent and/or subtasks: 3 days
3) attaching files: 1 week
4) attaching sketch: 1 week
5) documentation: 1 day

Contribution of member 3: (Edna Mutapcic)
My responsibility was creating notifications for the app as well as ensuring user can set preferences about notifications should be received. I used the strategy pattern to implement different notification types, pop up or email.

I did try to use use the JavaMail API for sending an email. However I had issues with adding JavaMail library to our app's dependencies, I spent 3 days trying to make it work. Unfortunately I had to give up on this idea, and decided to just show a confirmation taht email was sent, and store in the log. My part also was depending on some of the other member’s work so I had to wait and then familiar myself with the code as there were some changes in the comparison to DESIGN. I wanted to combine my part as best as possible with the rest of the code. So I decided to implement Observer for upcoming appointment notifications where I found the opportunity to use some of the existing data. Since I needed list of appointment tasks I decided that class which already contained this list and is updated regularly should be my concrete subject. I spent a few days figuring out the connection to make it work. I also implemented logic for checking if the upcoming appointment notification should be shown and used alert notification for this. I created notification preferences logic - saving and getting preferences, and also the xml file with the checkboxes which user can use to select which notifications should be received. For each action (create, update and delete) I added conditions to check which preferences are currently selected and based on that to send a notification. Created some JavaDoc comments All functionalities from Member 3 responsibility are implemented and strategy and observer patterns are implemented.

Contribution of member 4: (Mohammad Mahdi Fallah)

Implementation of fragments and switch tab : 1 day
implementation of list view and its adapters (using template method): 3 days
implementation of calendar view and its adapters (using template method): 2 days
implementation of hiding functionality and expanding observers/database/list view with persistence: 2 days
implementation of theme switch with persistence and expanding observer/database (using decorator): 2 days
implementing unit tests: 1 day
writing doc and recording video : 1 day

I have implemented both design patterns assigned to me namely, template method and decorator
you can find my implemented design patterns in view package under viewsetuptemplate package, and decorator pattern also in view under decorators package. there were added functions/expansions to other parts of the app to achieve my tasks which can be find in model under setting data package, also Task class under taskdata package, and all of their related daos/queries/observers to interact with view in viewmodel package I spent the last two days for implementing unit test and writing the documentation

Member 5 (Sahel Naderi):
Responsible for Import and Export of Tasks.

4.3 How-To Documentation

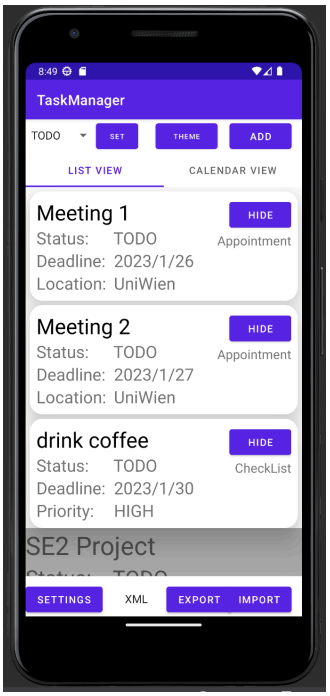
-first clone the directory from git by running :
git clone <the ssh or http url>

-then go to your chosen local directory and open SE2_TEAM_0206 folder, there you can find the working apk and reports, diagrams etc.

-by sending the apk to your android device you can install the app and use it on your physical device.

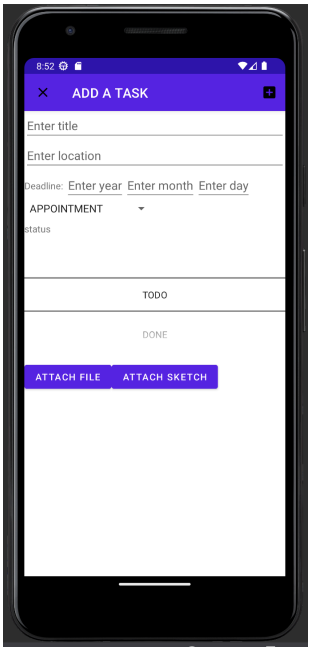
-for building the program on your local machine you should open the taskManager using your android studio and let the gradle to build.

after running the app on your desired emulator you are greeted with the first page of the app and some predefined task we created for you for testing

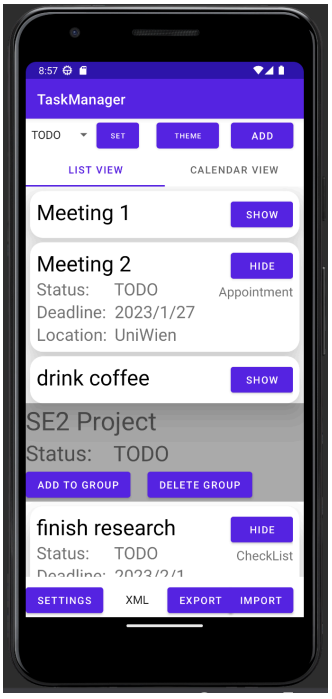


To add a task please click on the add button on the top right side of the window and you will be redirected to adding task page

fill out the fields by your desired input and click on the add button on top right to add the task.



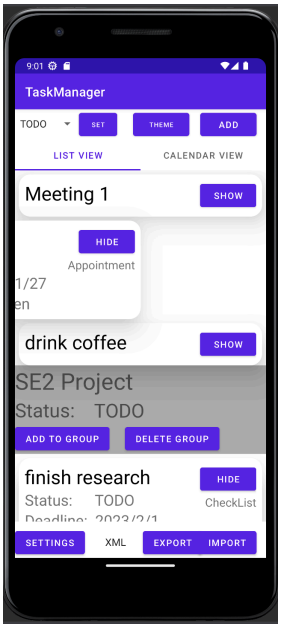
after task is added you will be getting a notification based on your preferences.
you can edit a task by touching it on the **List View** and you will get adding/editing task activity with all the fields filled with the chosen task.
touch the add button to save your changes



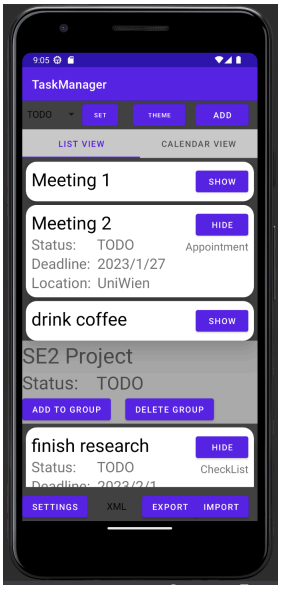
.
you can also interact with the **ListView** tasks by hiding not desired info of a task and keeping the title showing only by clicking the hide button (note that this is persisted to the data base and on your next start up the state of the tasks are loaded as you left them)

you can add checklist items to a group or delete them completely by touching their respective button on the group shown with darker tone.

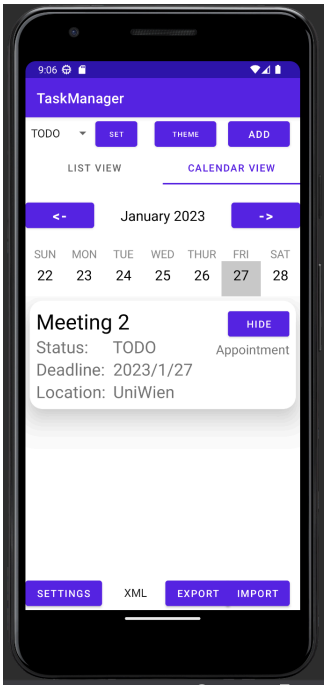
to delete a single task hold your finger on a task for a moment and swipe it to the left



by clicking on theme button you can switch theme which is persisted



you can switch to calendar view to see the tasks filtered by date:



choose your desired input for all tasks on the drop down on top left and click on set to change status of all of them.