Software Engineering 2

DESIGN REPORT

Team number:	0308

Team member 1	
Name:	Ilinca Vultur
Student ID:	11925311
E-mail address:	a11925311@unet.univie.ac.at

Team member 2	
Name:	Daria Lazepko
Student ID:	11826680
E-mail address:	a11826680@unet.univie.ac.at

Team member 3	
Name:	Jin-Jin Lee
Student ID:	11913405
E-mail address:	a11913405@unet.univie.ac.at

Team member 4	
Name:	Vladislav Mazurov
Student ID:	11710356
E-mail address:	a11710356@unet.univie.ac.at

Team member 5	
Name:	Miruna-Diana Jarda
Student ID:	11921801
E-mail address:	a11921801@univie.ac.at

1 Design Draft

1.1 Design Approach and Overview

We started our design approach by each member creating their own prototype in order for everyone to familiarize themselves with the project and the requirements. After this first phase, we evaluated all prototypes and discussed what the best solution would be and combined everything in one final prototype version. This visualization gave us a basic understanding of how we imagined the app could possibly be structured, what components we needed and at the same time it was also a common base so everyone was on the same page about how the app would look like.

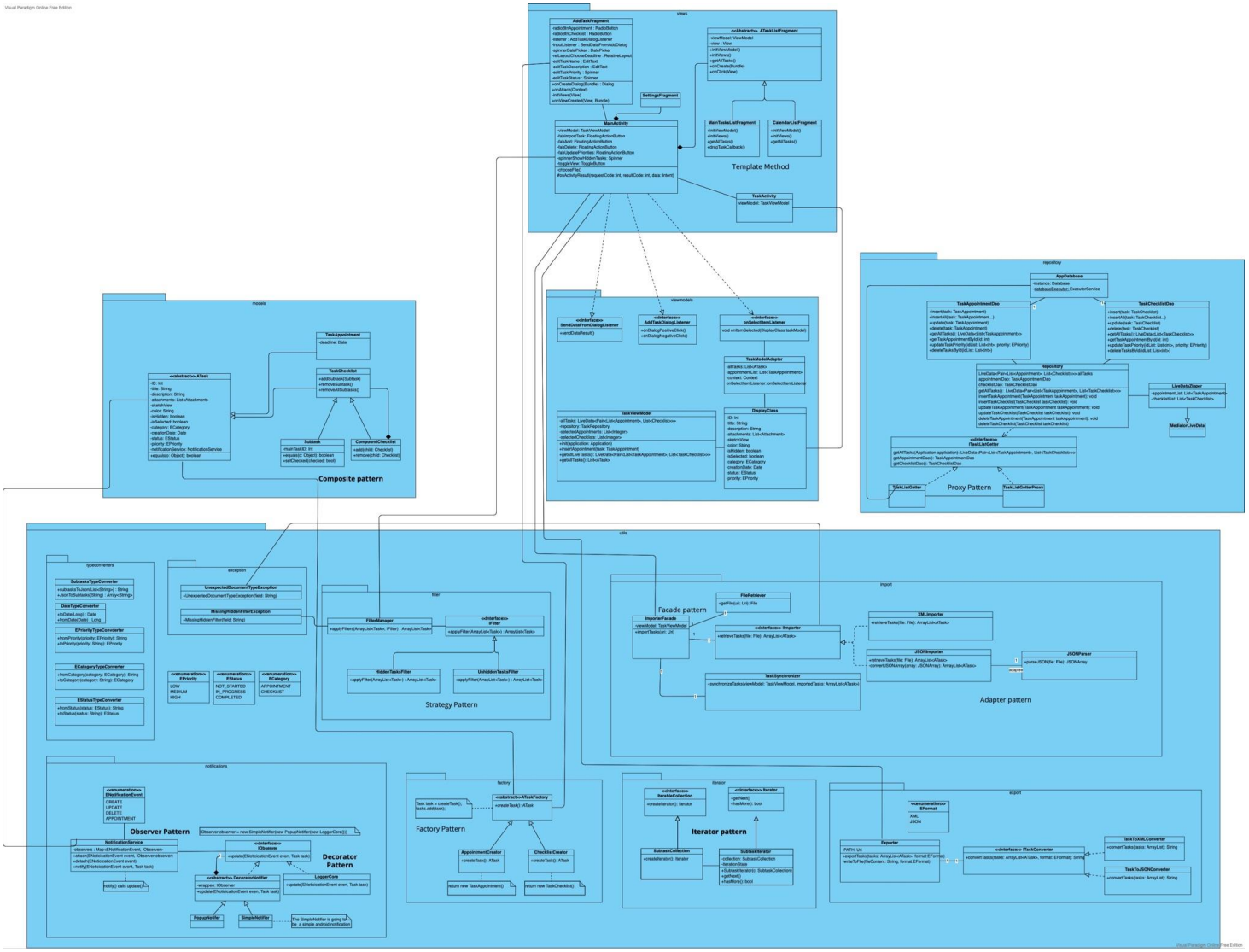
Afterwards, everyone studied all design patterns and we started to think about which design pattern could fit into which part of our application. At this point we did not assign individual member responsibilities yet because we thought that it was important for everyone to think about the general structure ´to get as many ideas as possible and not have everyone isolate themselves which could potentially lead to misfitting designs.This phase consumed most of our time and took place during the majority of our weekly meetings.

Towards the end of the design phase, after numerous iterations, we finally found a place for each pattern and decided to split up the member responsibilities so each member could focus and refine their respective parts.

For the coding of the basic functionality, we all set together to code and finish everything.

At the end, before the submission, we met once again to go through all parts of our design and fix still existing uncertainties and inconsistencies. This last meeting finalized our design and was also the day we finished writing the design report.

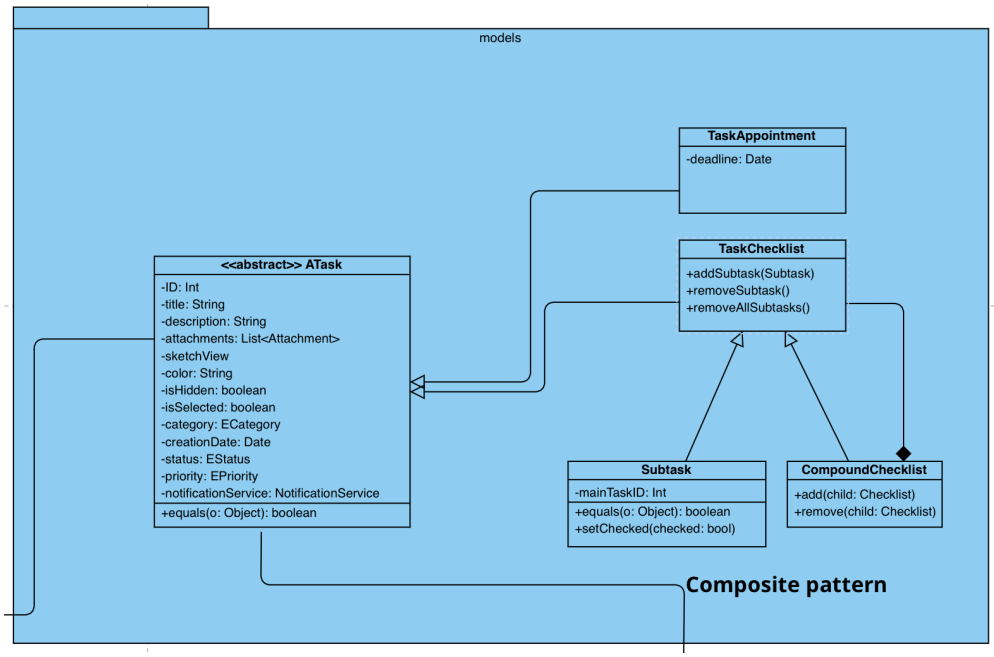
Overall Architecture



1.1.1 Class Diagrams

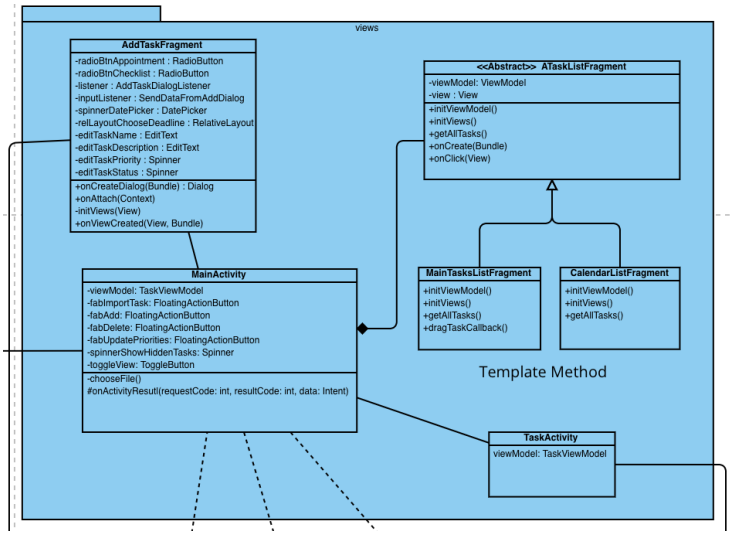
1.1.1.1 MODELS

The models package contains our data objects: the abstract class ATask, the 2 subclasses which represent our 2 task types TaskAppointment and TaskChecklist, as well as the Subtask class which is part of the Composite Pattern.



1.1.1.2 VIEWS & VIEWMODEL

We will have roughly 5 views:



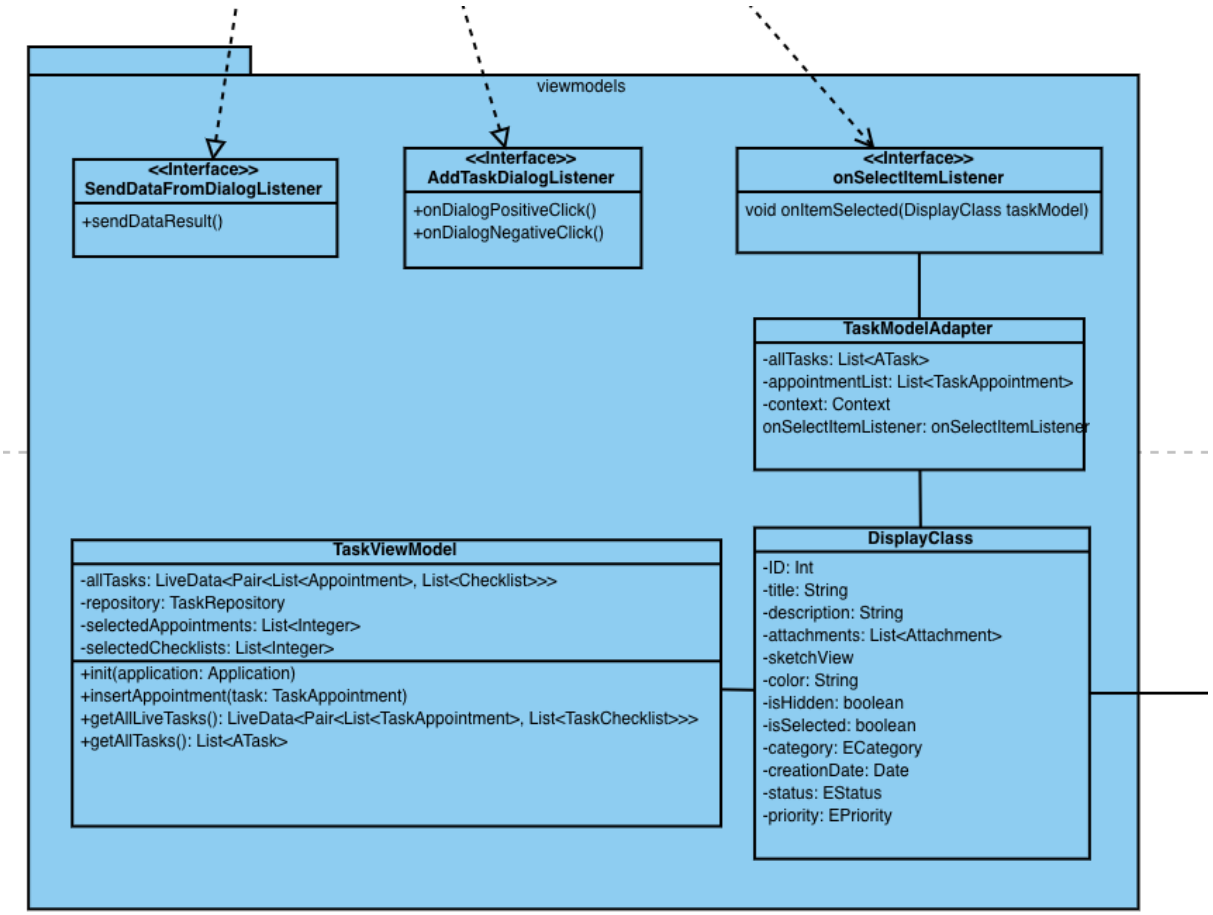
We want to implement the two views (*List* and *Calendar*) as two fragments that are contained in the MainActivity. There will be a toggle button to change between these 2 views in the main Activity.

The List fragment holds the RecyclerView that presents the list of tasks with both types present. The distinction between the two types will be made through an icon. We will have 3 FloatingActionButtons on this view: one for the deletion of the selected items, one for adding a new task and one for updating a common property on the selected

tasks (priority). We will also have a Spinner so that the user can choose between Show Hidden Tasks and Do Not Show Hidden Tasks.

The Calendar fragment will show a calendar: when the user presses on a date, the tasks of type appointment that have a deadline on that specific date will be shown underneath.

The *MainActivity* will implement two interfaces: SendDataFromDialogListener which handles the data that the user has entered in the AddTaskDialogFragment, and AddTaskDialogListener which handles the actions that take place when the user clicks on the dialog positive or negative buttons.



When the user clicks on the add new task button, an *AddTaskFragment* opens. This extends the DialogFragment from Android.

When the user clicks on one of the tasks in the list, *TaskActivity* opens and the details of the task appear. In this same view, the user can edit the details of the task and update it in the database by clicking the button Update.

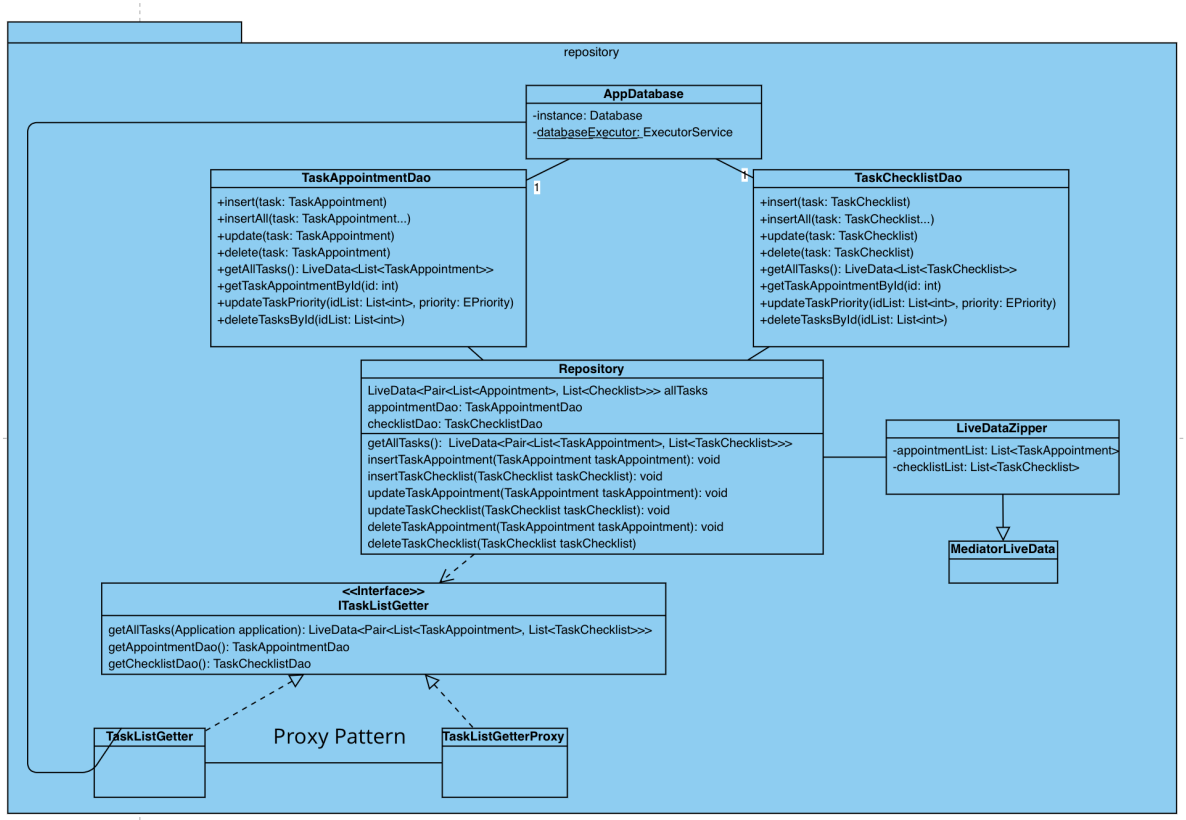
In the ViewModel package, we will implement the ViewModel and the Adapter for the RecyclerView. The ViewModel holds reference to the Repository and to the LiveData that comes from it and makes sure to update the view accordingly.
The Adapter implements all the logic needed to populate the RecyclerView with task items and also an interface onSelectItemListener so that the ViewModel can be updated accordingly when a user selects a task.

1.1.1.3 REPOSITORY

Repository package holds the database needed for the app. We use Room library to easily and persistently save the tasks and retrieve them.

We have AppDatabase which holds an instance of the whole database and two Dao classes needed by the Room library, each for different task types. There we manipulate data including insert, deletion, updates and getting all tasks, which all methods could be accessed through Repository class in the end.

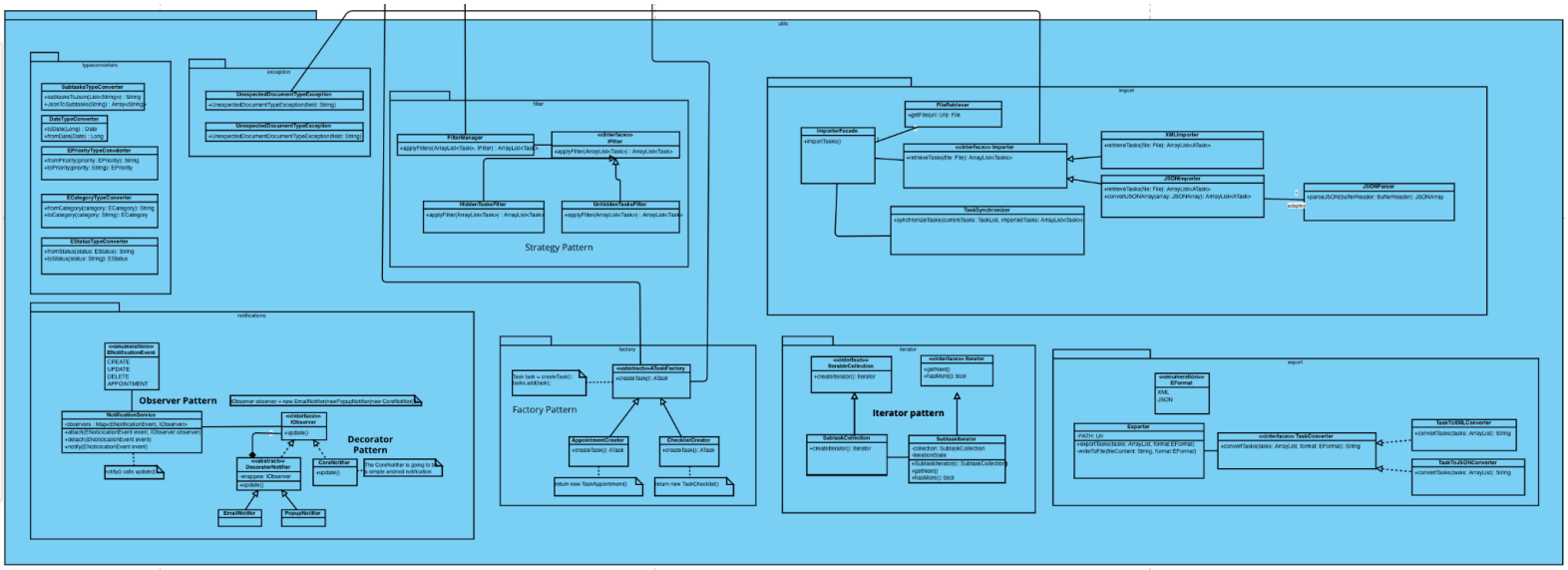
Additionally we decided to use Proxy pattern to access only necessary data and also not access databases each time we want to view tasks in the app.



1.1.1.4 UTILS

Utils package holds all the necessary “middle-man” classes and packages to function smoothly and in an understandable way.

We implemented type converters package not to overwhelm our other classes with converting types of enums, dates and subtasks. Filter package implements Strategy pattern specifically to filter through hidden/unhidden tasks on our main screen with the help of FilterManager. Notification package implements helper classes to show user desirable notifications in the up. Additionally it makes use of Observer and Decorator patterns. Factory package, hence the name, implements Factory pattern for creation of new tasks. The Iterator package will help us iterate through subtasks and use the Iterator pattern to do it. Additionally we have two packages for importing and exporting tasks in the app. Both make use of Adapter pattern and import package additionally implements Facade pattern. Import package will take JSON or XML files, convert it and import into the list, also synchronization is supported if the task already exists in the database. Export package does a reverse job and converts existing tasks to JSON/XML for the user to download.



1.1.2 Technology Stack

Minimum SDK:

API Level 21

Virtual Device:

We tested on both Pixel 4, Release: Q (Api 29) and Pixel 4, Release: R (Api 30).

Libraries:

Room:

website: https://developer.android.com/jetpack/androidx/releases/room?gclid=Cj0KCQiAj4ecBhD3ARIsAM4Q_jGn670tvLwAkIp1XX6o5l8sv1ixBL-FdfQWmwWxw0FWM0J4CKc1j10aAv7nEALw_wcB&gclsrc=aw.ds

version: 2.4.3

purpose: to be able to store our data in a local database

Material Design:

website: https://m2.material.io/develop/android/docs/getting-started

version: 1.7.0

purpose: to style the views

Gson:

website: https://github.com/google/gson

version: 2.10

purpose: to convert ArrayLists to a suitable data type (String), to be able to store it in the database

JUnit:

website: https://junit.org/junit4/

version: 4.13.2

purpose: Unit Testing

Mockito:

website: https://site.mockito.org/

version: 4.9.0

purpose: To mock functionality that is yet to be implemented but needed for unit testing

XStream:

website: https://x-stream.github.io/

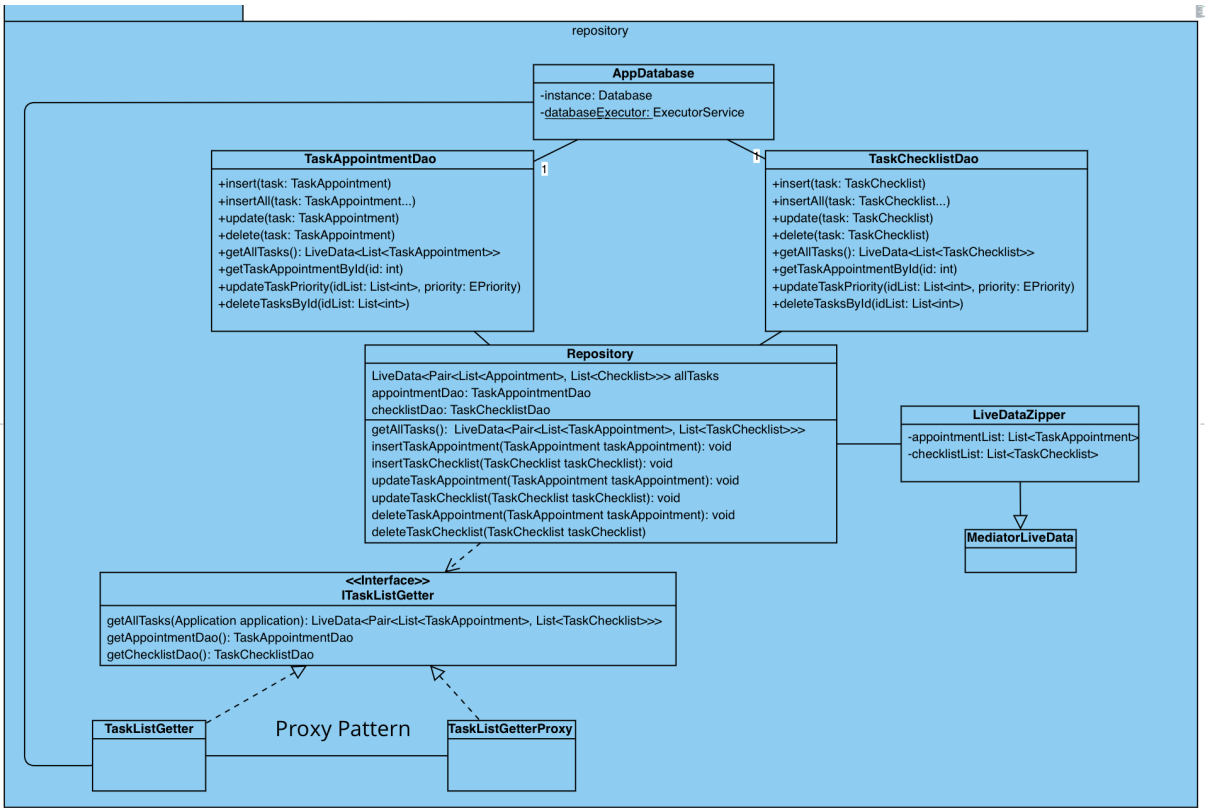
version: 1.4.12

purpose: To serialize and deserialize tasks to XML and the other way around

1.2 Design Patterns

1.2.1 Proxy Pattern

The Proxy Design Pattern can be used when we want to postpone the initialisation of an object to take place only when that object is needed in the app, so that we save up some resources.
We want to use the Proxy Pattern into our repository package in order to get rid of the need to always access the database in order to get the list of tasks for the recyclerview, since this would be a costly operation if done all the time. We do it one time in the beginning, when the list is null, we save it into our repository, and after that we work with the “proxy” list. To save both types of tasks into our repository. We are going to use a Zipper class which combines two LiveData objects that have different types together.



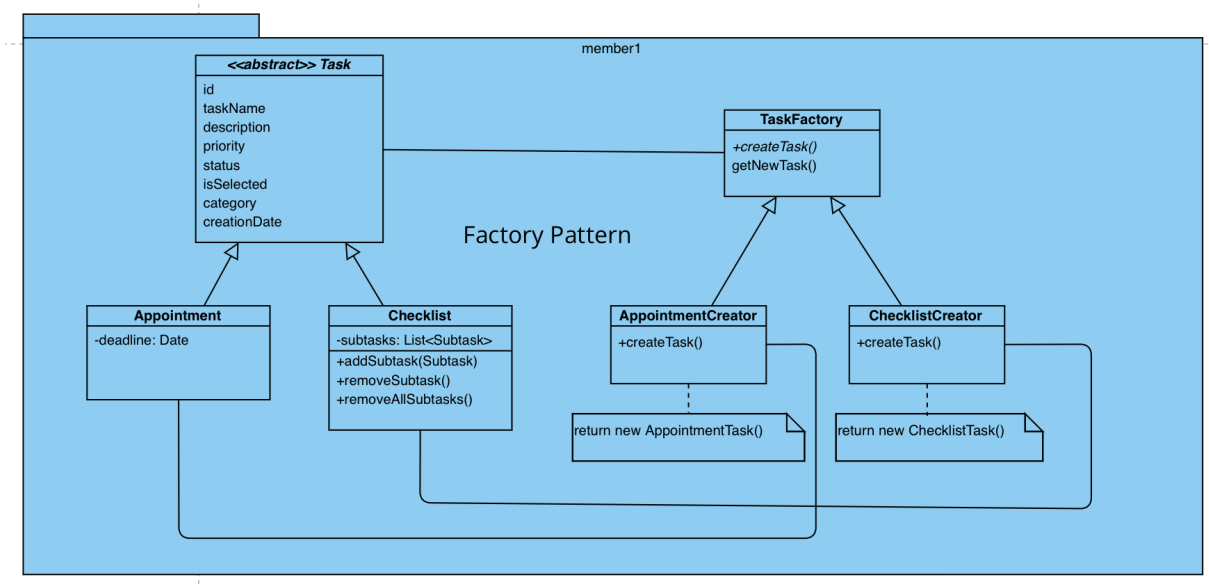
```java
public TaskRepository(Application application) {
    TaskList taskListProxy = new TaskListProxy();
    this.allTasks = taskListProxy.getAllTasks(application);
    taskAppointmentDao = taskListProxy.getAppointmentDao();
    taskChecklistDao = taskListProxy.getChecklistDao();
}
```

```java
public class TaskListProxy implements TaskList {

    private TaskList taskList;

    @Override
    public LiveData<Pair<List<TaskAppointment>, List<TaskChecklist>>> getAllTasks(Application application) {
        if (taskList == null) {
            taskList = new TaskListImplementation();
        }
        return taskList.getAllTasks(application);
    }
```

```java
public class TaskListImplementation implements TaskList {

    private static TaskAppointmentDao appointmentDao;
    private static TaskChecklistDao checklistDao;

    public static final String TAG = "TaskListImpl";

    @Override
    public LiveData<Pair<List<TaskAppointment>, List<TaskChecklist>>> getAllTasks(Application application) {
        AppDatabase database = AppDatabase.getDatabase(application);
        appointmentDao = database.taskAppointmentDao();
        checklistDao = database.taskChecklistDao();
        return new CombinedLiveData(appointmentDao.getAllTasks(), checklistDao.getAllTasks());
    }
```

1.2.2 Factory Pattern

The Factory Pattern is used when we want to hide the creational logic from the user and defer it to the respective subclasses.

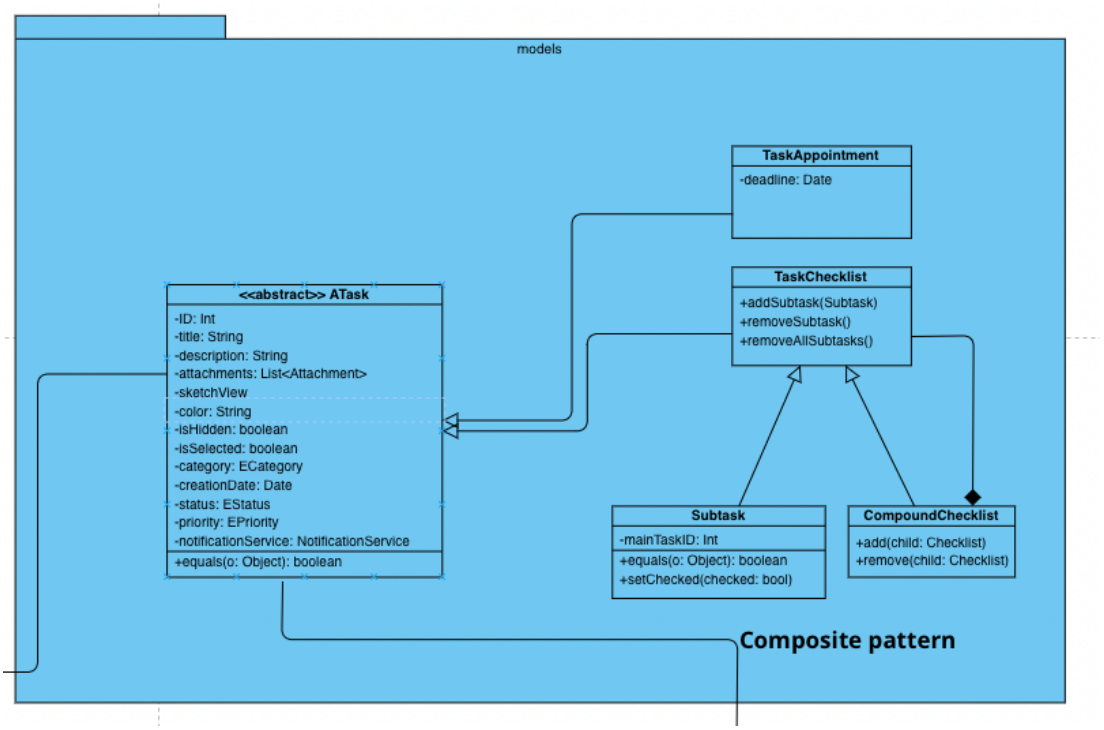


We will use the Factory Pattern for the creation of two different types of Tasks: Appointment and Checklist. When adding a new task, a user will be able to select which type of task they want to add. This is where it will be decided which type of object needs to be created. The abstract class Task will own the common properties of both types.

```
if (isSelectedAppointment) {
    taskFactory = new TaskAppointmentFactory();
    viewModel.insertAppointment((TaskAppointment) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, new ArrayList<>()));
} else {
    taskFactory = new TaskChecklistFactory();
      viewModel.insertChecklist((TaskChecklist) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, new ArrayList<>()));
}
```

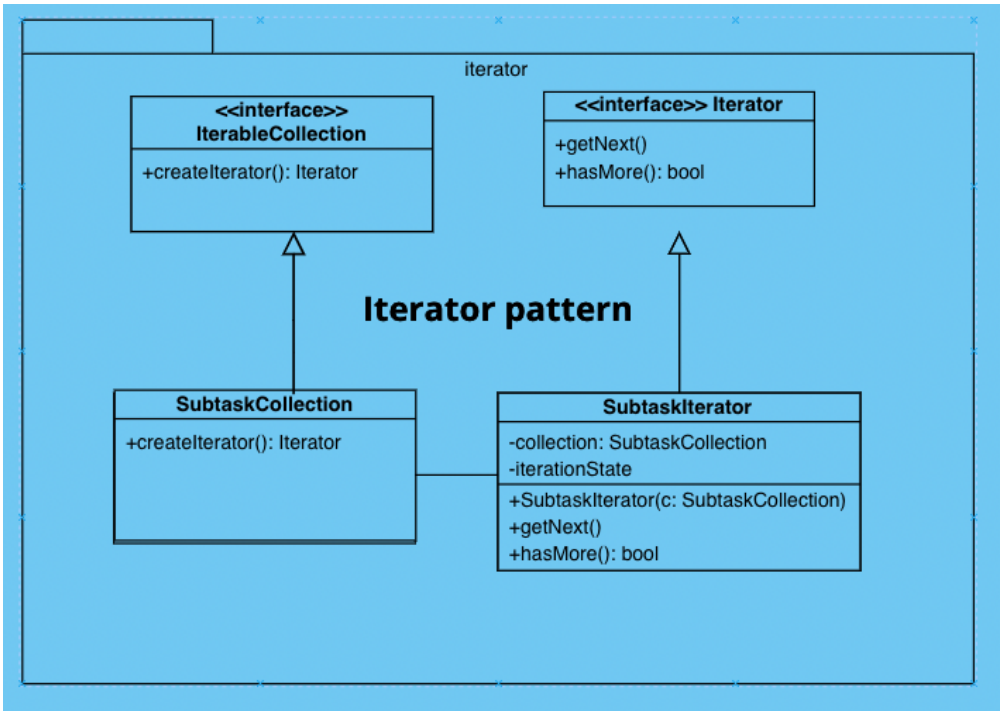
1.2.3 Composite Pattern

Composite is a structural pattern which is mainly used to work with tree object structures in the code. The basic idea is to have two element types: leaves and containers - which share the same interface. A container stores a list of both element types. In this way a nested recursive object structure can be represented.
We decided to use this pattern to represent the nested tasks structure. In our case only the checklist task can contain subtasks and each subtask can contain more subtasks. However, the tree level will be limited so the user would not be able to create infinite levels of subtasks.
Remark: The pattern is planned to be implemented for the final phase as a member responsibility so there are no code snippets available yet.



1.2.4 Iterator Pattern

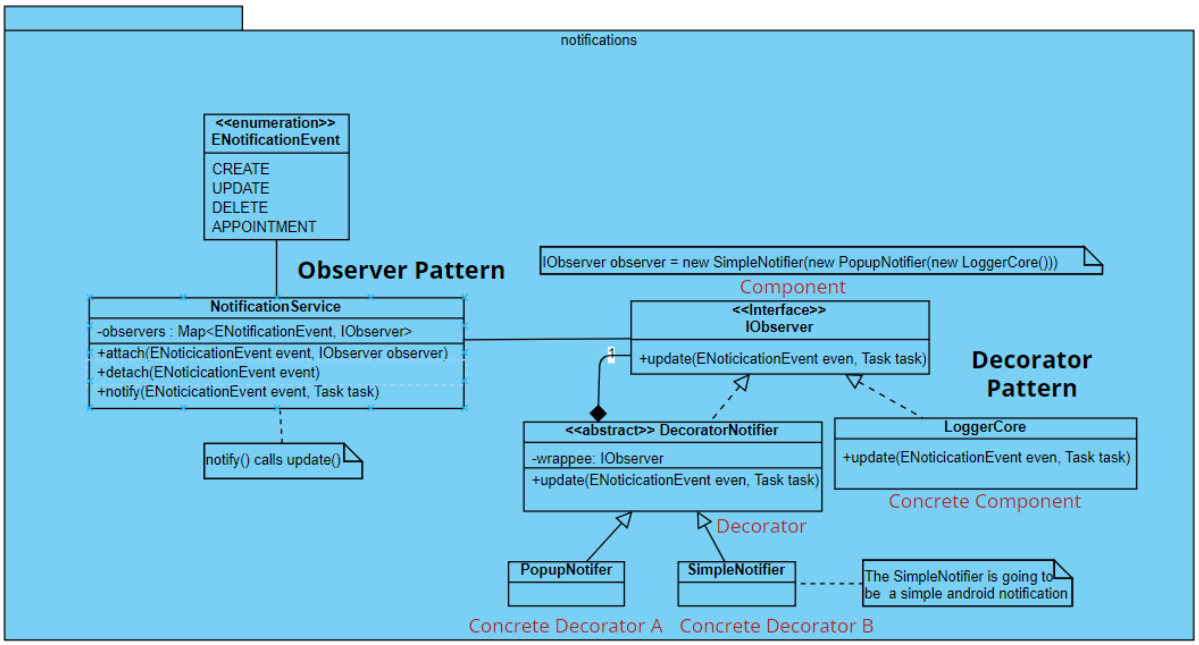
Iterator is a behavioral pattern which is used to traverse a collection of elements without exposing its internal representation. It is often used combined with the composite pattern. So we decided to use it to traverse the subtask structure (see the diagram above).
Remark: The pattern is planned to be implemented for the final phase as a member responsibility so there are no code snippets available yet.



1.2.5 Decorator Pattern

The decorator pattern is a structural design pattern that allows users to dynamically attach new functionality to an object without altering its structure. This is accomplished by recursively wrapping a core object in so-called decorators that contain the additional behaviors. It is a flexible alternative to subclassing where whatever combination of behaviors can be specified without having to create a new class that comprises all behaviors for each combination.

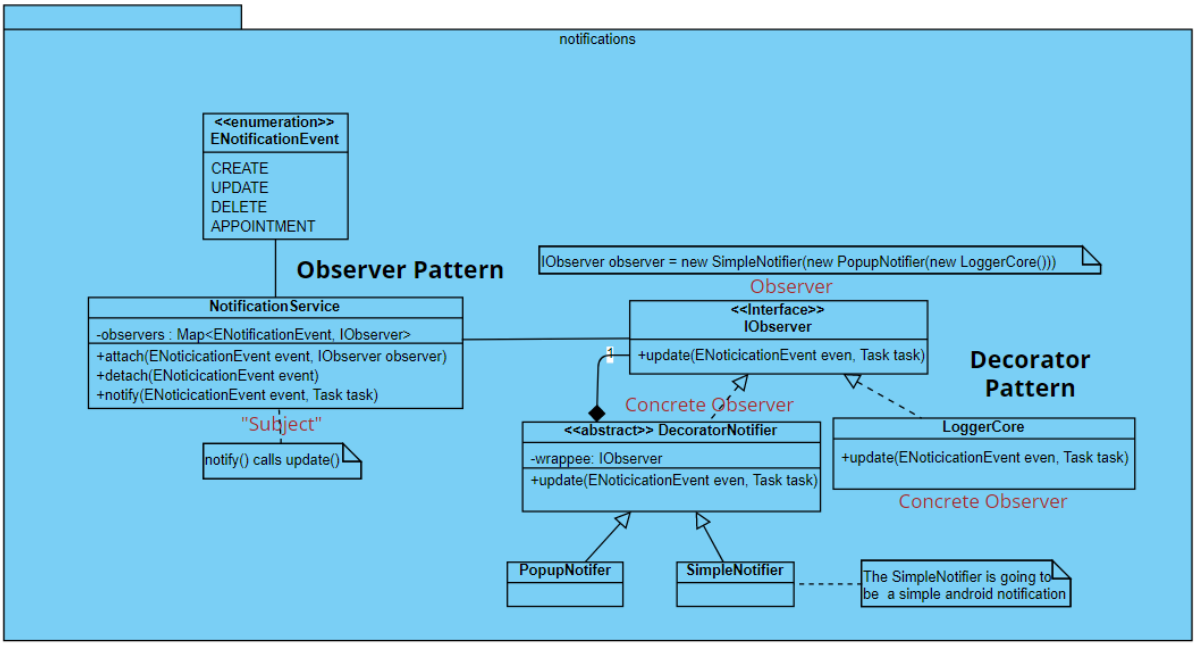
We decided to use the decorator pattern for our notification system (member 3 responsibility). The users can dynamically select how they want to be notified about events, whether by a simple android notification or popup or even both. Depending on the selection, we build a notifier using the decorator pattern with a logger as its core functionality and popup and simple notification as the decorators (optional wrappers) that contain the additional behaviors.
If we were to use subclassing to realize our notification system then we would have had to create a new class for each new combination of notification types, e.g. new class PopupAndSimpleNotifier if we wanted popup and simple notification at the same time. With more notification types and endless combinations, it is nonsensical to create a new class each time, that is why we opted to use the decorator here. With the decorator we will be able to dynamically add and remove notification types to whatever combination we want while adhering to the single responsibility and open–close principle



1.2.6 Observer Pattern

The observer pattern is a structural design pattern that defines a one-to-many relationship between an observable and multiple observers, where the observable notifies the observers about changes in its state.

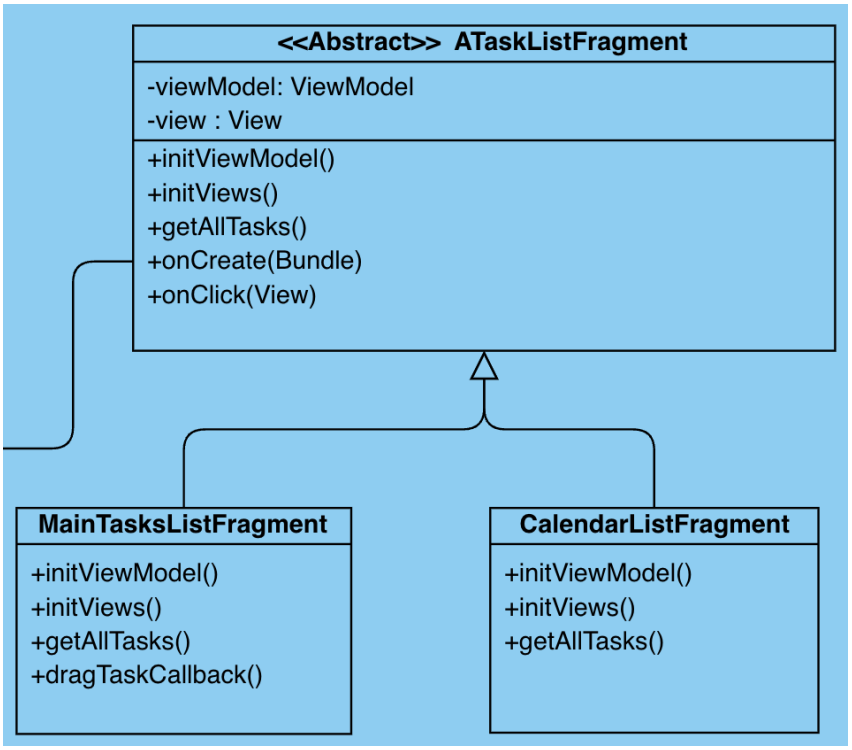
We decided to also use the observer pattern for our notification system (member 3 responsibility). The users can dynamically select from a list of actions (onCreate, onUpdate and onDelete) what they want to be notified by and depending on their selection we attach/detach observers. The observable object in our case would be the task and the observer is an IObserver (also part of the decorator) that sends out the notifications. However, because of the single responsibility principle, we separated the notification functions from the task as its own class NotificationSystem and the task holds an instance of NotificationSystem which holds a map with the notification event and the corresponding observer.
Without the observer pattern the observers would have to constantly inquire about the state of the observable which would be really resource intensive. This drawback is one reason why we use the observer pattern for the notifications.



1.2.7 Template Pattern

Template Method defines some skeleton class with some steps in it which other derived subclasses can override for their need without changing its structure.

We decided to use this pattern to show two different views for normal tasks list and calendar view. An abstract class for a Fragment will hold basic functionality of showing it in the MainActivity. Derived classes for

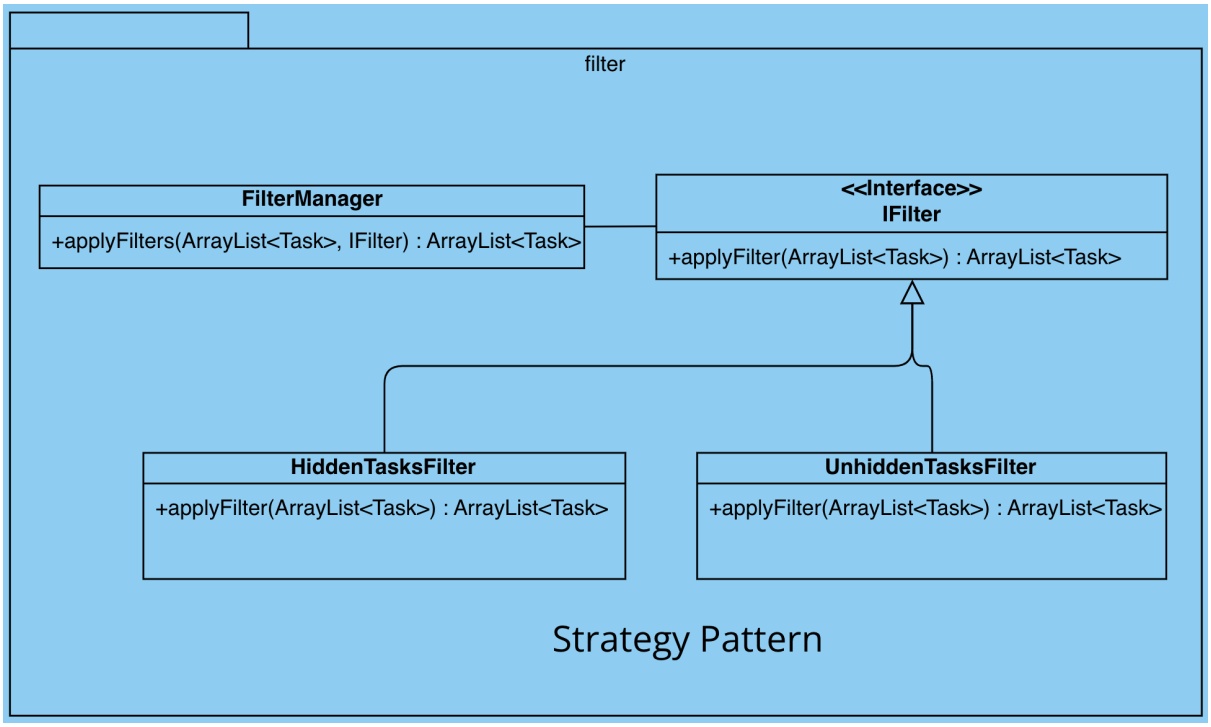


tasks list and calendar view implements necessary adjustment to show them differently based on the selected view mode.

1.2.8 Strategy Pattern

Strategy is a pattern that does some specific task in different ways. Meaning that we can extend some abstract class with different algorithms which will bring us to the same goal.

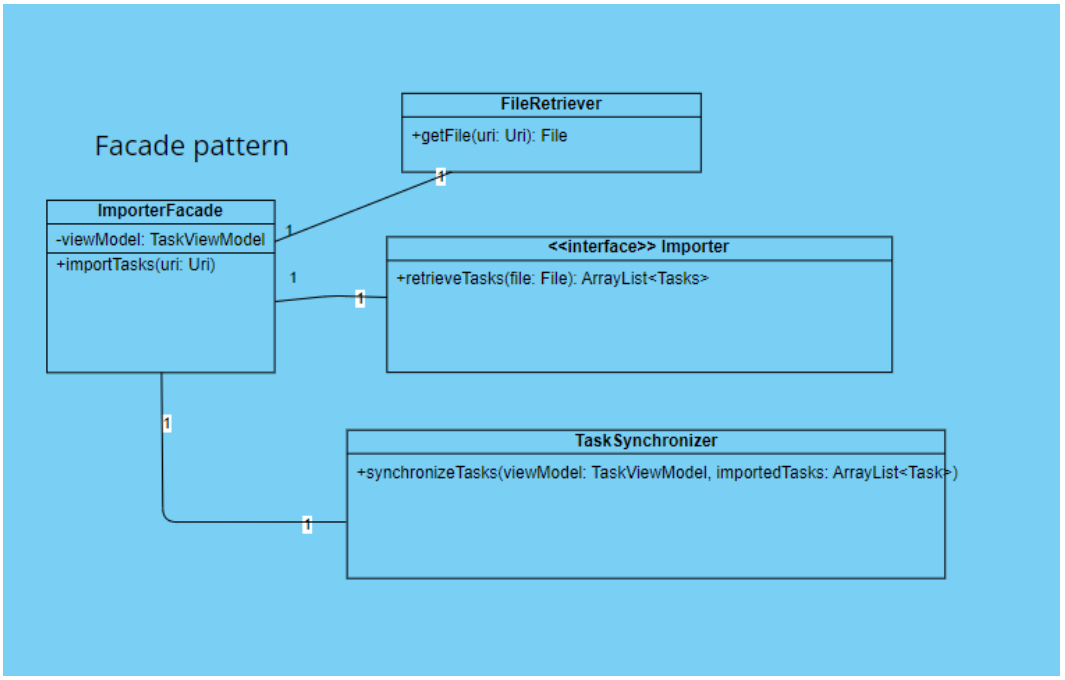
We decided to use this pattern for filtering the tasks as we need to show hidden and unhidden tasks on the main view. With the help of FilterManager and IFilter interface we can switch between those two tasks easily and smoothly. Additionally implementation will be scalable for additional filters if needed.



1.2.9 Facade Pattern

The facade pattern allows unifying behaviour of multiple entities while hiding their implementation from the client in order to simplify their use and facilitate loose coupling.

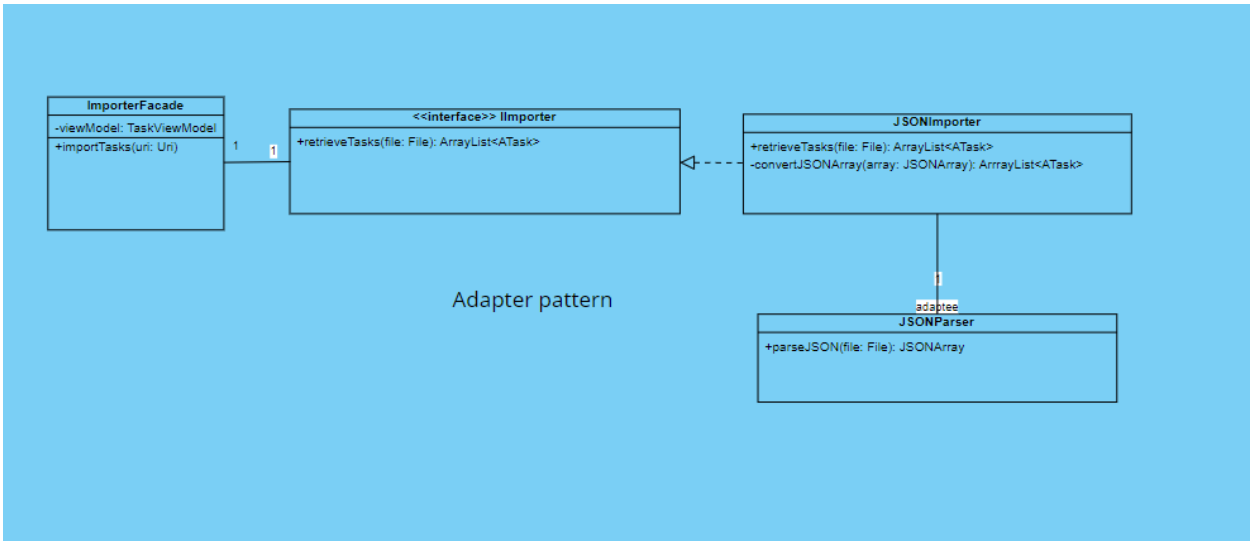
In order to import the tasks we use Intent.ACTION_GET_CONTENT in the chooseFile() method in the MainActivity to get the UR. Once we get this we use the ImporterFacade (with the TaskViewModel passed as an argument to the constructor) to actually import the tasks. The import is divided in 3 parts: first, we retrieve the file (this class can be reused for the file attachment responsibility of member 2), then we actually retrieve the content of the file and we parse it accordingly to get the tasks saved in that file, then we use the Task Synchronizer to compare the tasks currently in the repository in the view model with the newly imported tasks, and then ultimately update the repository.



1.2.10 Adapter Pattern

The adapter pattern is used to facilitate two entities with incompatible interfaces to communicate with each other by wrapping the behaviour of the one we want to use with our client into the implementation of an abstract entity that works with our client.

In our case, the ImporterFacade expects to work with a list of ATasks, regardless of the format of the import file, that can be further passed to a class that does task synchronization. The JSONParser, however, returns a JSONArray object, which makes it incompatible with what ImporterFacade expects. Consequently, we wrap the JSONParse into the JSONImporter class where we convert from JSONArray to a list of tasks.



2 Code Metrics

We analyzed the code metrics using the default lint tool provided by the Android Studio. And we also installed an additional plugin - “Statistic” from IntelliJ - to better analyze other parameters, which were absent in lint, such as the number of code lines, classes and packages.
A short overview of the results:

Number of packages: 8
Number of classes: 31

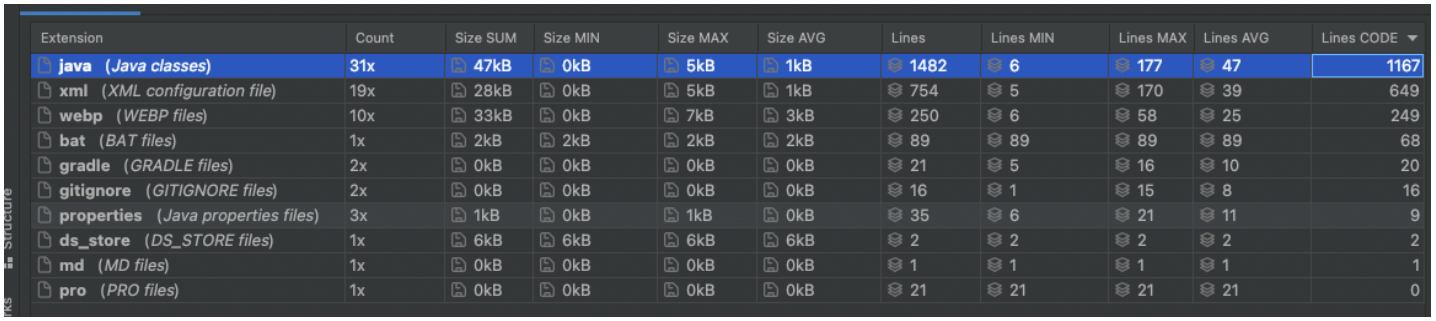
Lines of code

- Total: 2671
- In classes: 1167
- In XML: 649
- comments in classes: 12

Current bugs

During the development process and testing out the app on the emulator, only a few bugs were found on the current stage of the project.

- Validation: currently there is now validation implemented for the user input so e.g. a task with an empty title can be created
- lint statistics report exposed some minor performance issues, such as:
    - useless parent in *activity_task.xml* file;
    - in *MainActivity* and *TaskListAdapter*: improve efficiency by using specific change events instead of notifyDataSetChanged()



Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines	Lines MIN	Lines MAX	Lines AVG	Lines CODE
java (*Java classes*)	31x	47kB	0kB	5kB	1kB	1482	6	177	47	1167
xml (*XML configuration file*)	19x	28kB	0kB	5kB	1kB	754	5	170	39	649
webp (*WEBP files*)	10x	33kB	0kB	7kB	3kB	250	6	58	25	249
bat (*BAT files*)	1x	2kB	2kB	2kB	2kB	89	89	89	89	68
gradle (*GRADLE files*)	2x	0kB	0kB	0kB	0kB	21	5	16	10	20
gitignore (*GITIGNORE files*)	2x	0kB	0kB	0kB	0kB	16	1	15	8	16
properties (*Java properties files*)	3x	1kB	0kB	1kB	0kB	35	6	21	11	9
ds_store (*DS_STORE files*)	1x	6kB	6kB	6kB	6kB	2	2	2	2	2
md (*MD files*)	1x	0kB	0kB	0kB	0kB	1	1	1	1	1
pro (*PRO files*)	1x	0kB	0kB	0kB	0kB	21	21	21	21	0

3 Team Contribution

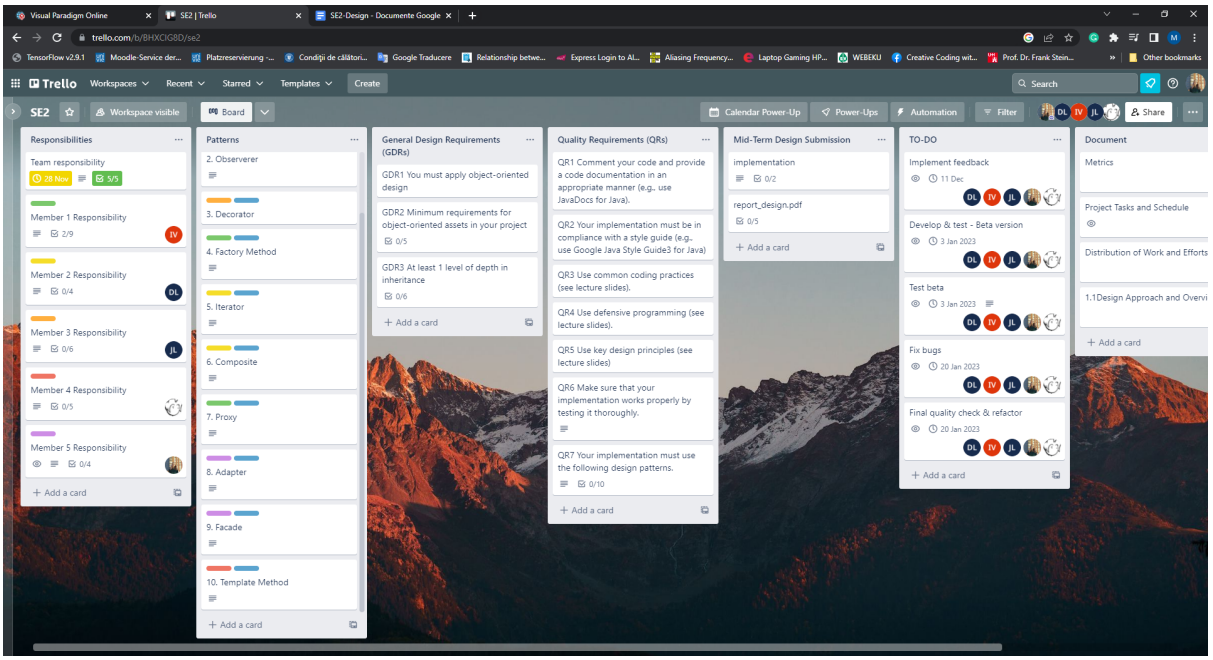
3.1 Project Tasks and Schedule

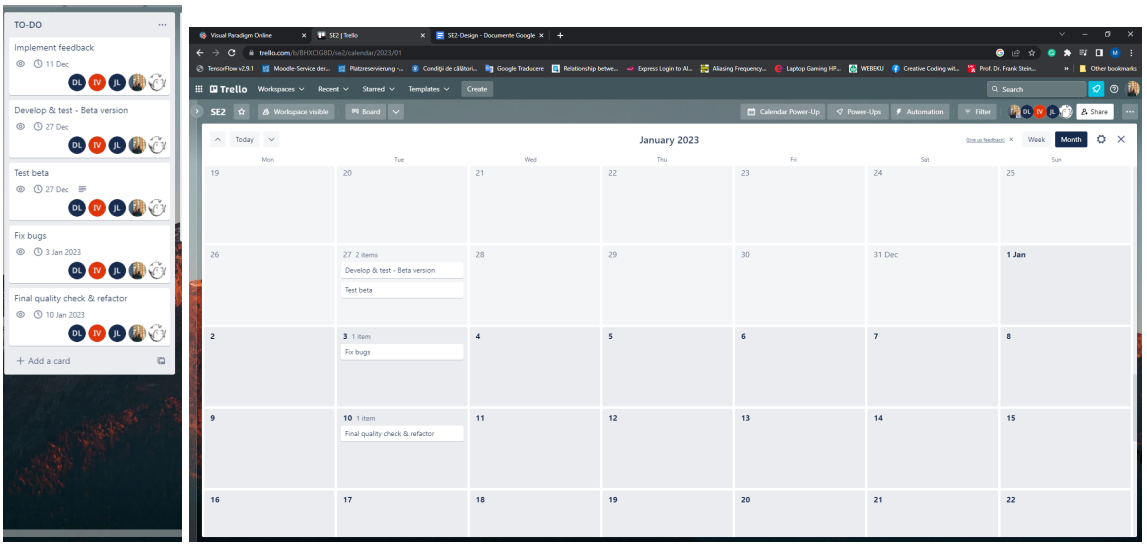
In order to have a good overview over the main requirements, details and deadlines of this project, we have used Trello from the very beginning of our group work.

Different aspects of the project were organized in different lists. For example, to make sure we have covered all patterns and that each member implements at least two patterns, we have created a list that tracks whether we have made a final decision regarding where to use that certain pattern (blue) and which member implements which pattern (colour depends on the assigned member). We also use Trello as a reminder for the way we assigned the member responsibilities.

We have decided not to create a Gantt chart as it does not offer the flexibility offered by Trello. Instead, we have created the “TO DO” list in our workspace where we added the next steps of the project with deadlines, descriptions where needed and team mates assignment. This way, we have all the information needed for our project in one place and can easily adapt each step, deadline, member assignment, description and create checklists at the beginning of each step that would explain in detail what each member should do in that step. The calendar also allows us to see the deadlines for each step.

Our working methods involve testing while implementing the code and continuously reporting progress, difficulties we encounter and possible solutions to these. However, we will still have two sessions: one on the 27th of December and one on the 10th of January when we will go test everything together in order to identify possible problems we previously missed.


Trello
SE2
Responsibilities
Team responsibility
Member 1 Responsibility
Member 2 Responsibility
Member 3 Responsibility
Member 4 Responsibility
Member 5 Responsibility
Patterns
2. Observerer
3. Decorator
4. Factory Method
5. Iterator
6. Composite
7. Proxy
8. Adapter
9. Facade
10. Template Method
General Design Requirements (GDRs)
GDR1 You must apply object-oriented design
GDR2 Minimum requirements for object-oriented assets in your project
GDR3 At least 1 level of depth in inheritance
Quality Requirements (QRs)
QR1 Comment your code and provide a code documentation in an appropriate manner (e.g., use JavaDocs for Java).
QR2 Your implementation must be in compliance with a style guide (e.g., use Google Java Style Guide3 for Java)
QR3 Use common coding practices (see lecture slides).
QR4 Use defensive programming (see lecture slides).
QR5 Use key design principles (see lecture slides)
QR6 Make sure that your implementation works properly by testing it thoroughly.
QR7 Your implementation must use the following design patterns.
Mid-Term Design Submission
implementation
report_design.pdf
TO-DO
Implement feedback
Develop & test - Beta version
Test beta
Fix bugs
Final quality check & refactor
Document
Metrics
Project Tasks and Schedule
Distribution of Work and Efforts
1.1Design Approach and Overvi


TO-DO
Implement feedback
Develop & test - Beta version
Test beta
Fix bugs
Final quality check & refactor
January 2023

3.2 Distribution of Work and Efforts

Besides the common contributions that were made by all team members, we distinguish the individual contributions brought by each member:

Contribution of member 1 - Ilinca Vultur:

Member 1 was responsible for the implementation of the Factory and Proxy patterns, as well as with the implementation of the Adapter for the RecyclerView in the Main Activity and the implementation of the Main Activity itself. She also created the respective Main Activity Layout. In the FINAL phase of the project, member 1 has to add the remaining requirements of its responsibility (e.g. implement the second type of task, add the remaining properties, update the layout accordingly, etc.).

Contribution of member 2 - Daria Lazepko:

Member 2 was responsible for modeling of the Composite and Iterator patterns to further implement it in the final phase according to their responsibilities as well as one of the converters - SubtaskConverter. The member also implemented the tasks models and the converter for the subtasks. The member was also working on the design report on the following sections: 1.2.3, 1.2.4, code metrics and the project timeline.

Contribution of member 3 - Jin-Jin Lee:

Member 3 was responsible for modeling the observer and decorator pattern and will be responsible to further implement the patterns and their responsibilities in the final phase. The member implemented all enums, add_task_dialog_fragment and TaskActivity in the code. Additionally the member also worked on 1.2.5, 1.2.6 and 1.1 in the design report.

Contribution of member 4 - Vladislav Mazurov:

Member 4 was responsible for the implementation of Template Method and Strategy patterns. Both of those patterns will be implemented for the final phase of the project. This includes the whole filter package and necessary classes to show task list and calendar views. Additionally members implemented app database, task repository, date converter and fragment for adding the task.

Contribution of member 5 - Miruna-Diana Jarda:

Member 5 will implement the Adapter and the Facade patterns for the final phase. In the coding of the team responsibility, they were responsible for implementing the daos, the enumeration type converters, the TaskViewModel and one of the xml layouts. Furthermore, they keep the Trello workspace up-to-date.