Software Engineering 2
FINAL 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 Final Design

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 finished everything.

After we received feedback, we adjusted our design accordingly to comply with it. Each member started working on their part and member responsibilities as it was described in the design phase. When someone from the team had troubles then we tried to help and find a solution together, apart from that the app was constantly tested from user perspective which helped us find and fix bugs in each other's functionality. We also communicated to have a clear picture on which stage we are currently.

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

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. 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 needed for unit testing

XStream:

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

version: 1.4.12

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

ColorPicker:

website: https://github.com/duanhong169/ColorPicker

version: 1.1.6

purpose: To show color picker popup to change background color of tasks

1.2 Major Changes Compared to DESIGN

Member 1

I have kept most parts of my code the same as in the design phase. A major change that has affected my part was the migration from the MainActivity as the main point of the app to the abstract ATaskListFragment that serves as a base for both fragments ListFragment and CalendarFragment. In the design phase, some of my UML diagram representation names were not synchronised with the class names that I had in the code, so I changed those. I also had to modify some class types that were noted as abstract in the diagram but their representation didn't match the abstract type.

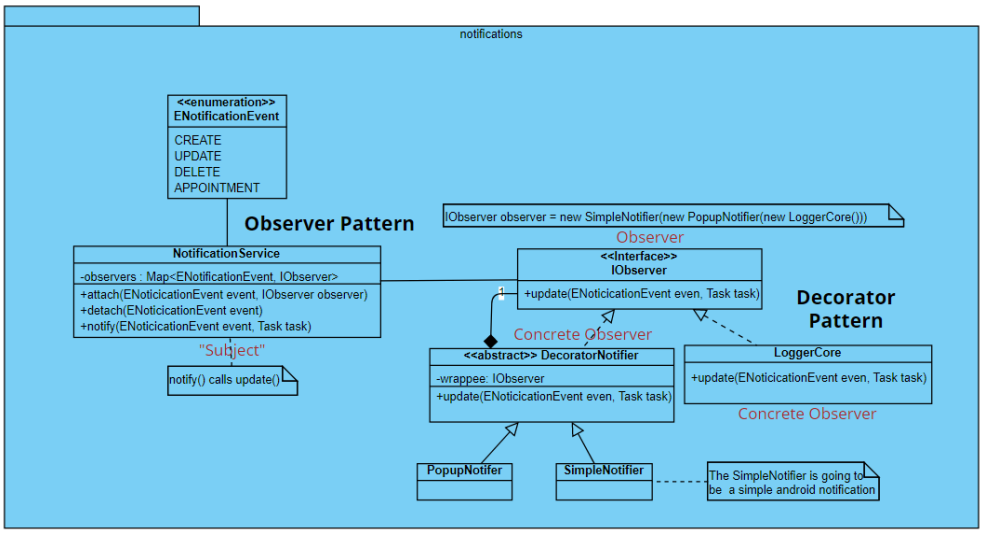
Member 2

There were some changes in realising the Composite Pattern. In the design phase the Subtask and CompountChecklist inherited TaskChecklist therefore also inherited all its properties, which we were not planning to have for the subtask. So I created an additional interface for the subtask. Also during the design phase it was not specified where the Iterator pattern for the subtasks should be applied. So I decided to use it when the state (if it's checked) of a parent subtask is changed and needs to be set for all children subtasks (which may also be nested).

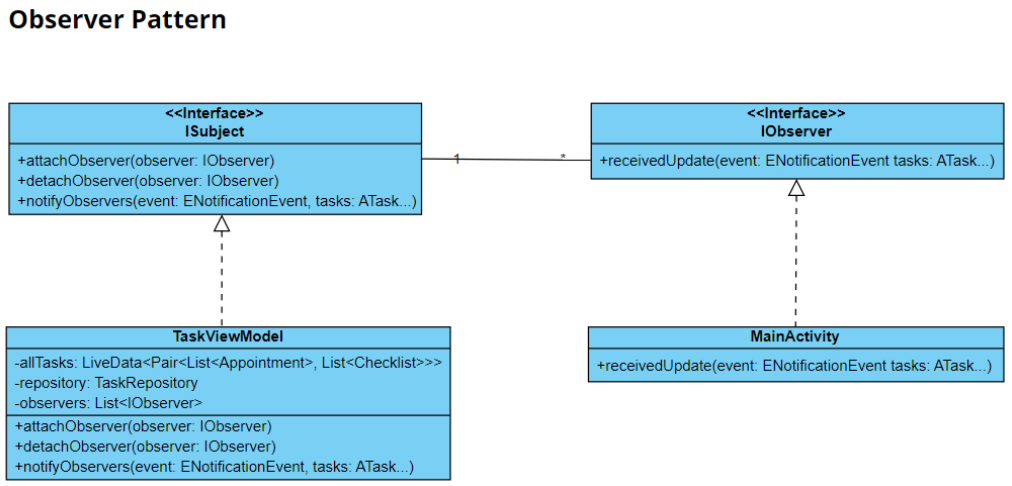
Member 3

The observer pattern has undergone a major change because I realised that making the task the observable subject and calling the notify method every time one of the fields was set, was not really feasible. The properties of a task are set by calling each setter individually and notifying the observers everytime was unnecessary. Instead it made more sense to make the TaskViewModel the subject and every time a task is created, updated or deleted, we call the notifyObserver() method and pass along the task that has changed. Because of this change a lot of class names were also changed.

Old:



New:



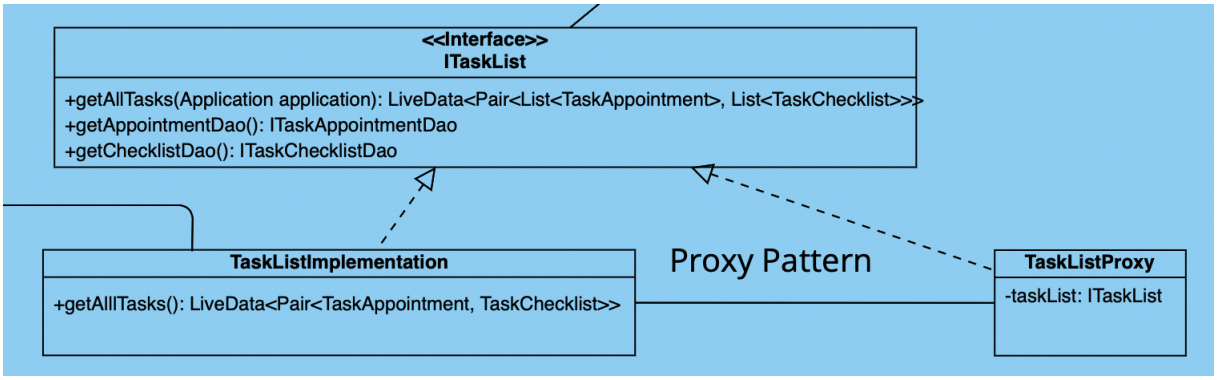
Member 4

For the design phase both strategy and template patterns were not in place. Therefore both of them were implemented for the final phase. Task list on the main screen were part of the main activity and were changed to fragment for the purpose of implementing template method and use of this list also for calendar view. Strategy pattern and filtering of the tasks based on isHidden value were also implemented for the final phase as changing the background color of the tasks and switching between app themes in the settings view.

1.3 Design Patterns

1.3.1 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 use the Proxy Pattern in 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 using a Zipper class called CombinedLiveData which combines two LiveData objects that have different types together.
The common interface that both real subject and proxy implement:

```
public interface ITaskList {
    LiveData<Pair<List<TaskAppointment>, List<TaskChecklist>>> getAllTasks(Application application);
    ITaskAppointmentDao getAppointmentDao();
    ITaskChecklistDao getChecklistDao();
}
```

The real subject (TaskListImplementation):

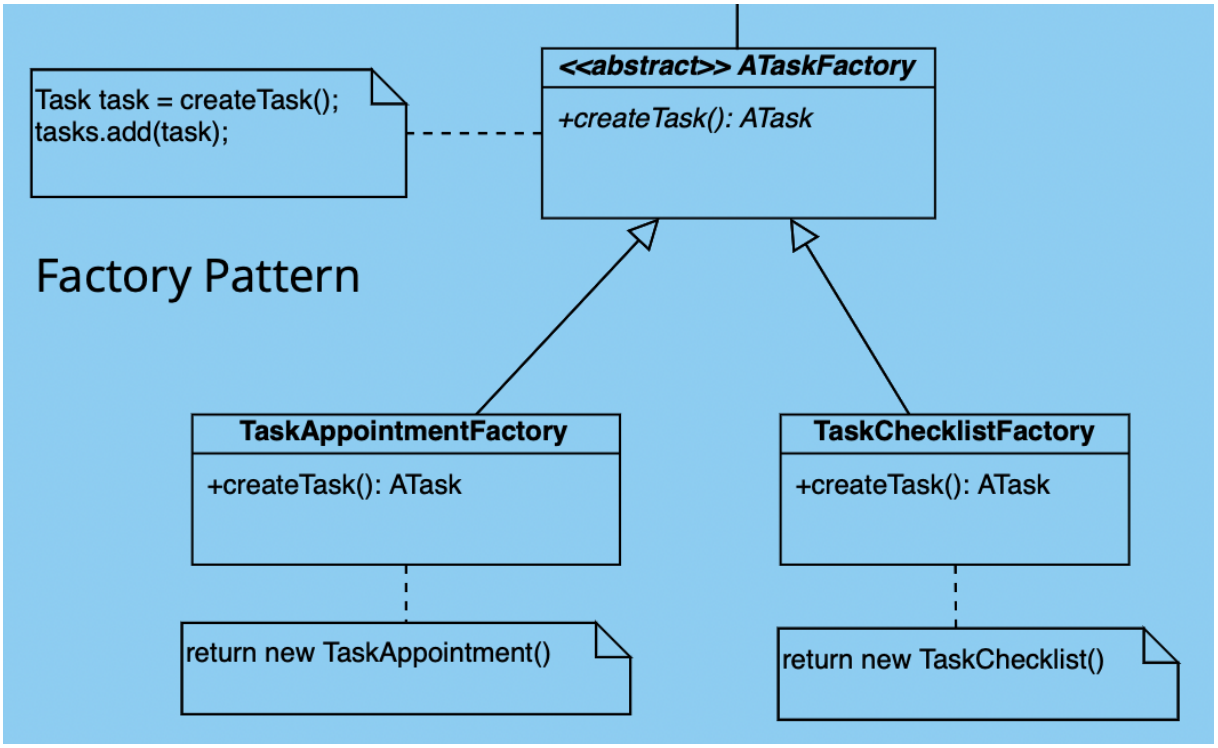
```
@Override
public LiveData<Pair<List<TaskAppointment>, List<TaskChecklist>>> getAllTasks(Application application) {
    try {
        AppDatabase database = AppDatabase.getDatabase(application);
        appointmentDao = database.taskAppointmentDao();
        checklistDao = database.taskChecklistDao();
    } catch (SingletonDbDoubleInitException e) {
        Log.d(TAG, e.toString());
    }

    return new CombinedLiveData(appointmentDao.getAllTasks(), checklistDao.getAllTasks());
}
```

The proxy (TaskListProxy):

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

1.3.2 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. This is needed so that the Open-Close Principle is respected.



We are using the Factory Pattern for the creation of two different types of Tasks: TaskAppointment and TaskChecklist. When adding a new task, a user is able to select which type of task they want to add. This is where it is decided which type of object needs to be created. The abstract class ATask will own the common properties of both types.
The abstract class factory from which all concrete factories inherit:

```
public abstract class ATaskFactory {
    public ATask getNewTask(String taskName, String description, EPriority priority, EStatus stat
        return createTask(taskName, description, priority, status, deadline, subtasks, attachment
    }
    public abstract ATask createTask(String taskName, String description, EPriority priority, ESt
}
```

Two concrete factories:

```
public class TaskAppointmentFactory extends ATaskFactory {

    @Override
    public ATask createTask(String taskName, String description, EPriority priority, EStatus status, Date
        return new TaskAppointment(taskName, description, priority, status, ECategory.APPOINTMENT, deadlir
    }
}
```

```
public class TaskChecklistFactory extends ATaskFactory {

    @Override
    public ATask createTask(String taskName, String description, EPriority priority, EStatus sta
        return new TaskChecklist(taskName, description, priority, status, ECategory.CHECKLIST, s
    }
}
```

Concrete Products:

```
@Entity(tableName = "task_appointments")
public class TaskAppointment extends ATask implements Parcelable {

    @TypeConverters(DateConverter.class)
    private Date deadline;
```

```
@Entity(tableName = "task_checklists")
public class TaskChecklist extends ATask implements Parcelable {
    @TypeConverters(SubtasksConverter.class)
    List<ASubtask> subtasks;
```

Abstract Product:

```
public abstract class ATask {
    public static final String TAG ="Task";

    @PrimaryKey(autoGenerate = true)
    private int id;
    private String taskName;
    private String description;
    private EPriority priority;
    private EStatus status;
    private boolean isSelected = false;
    private ECategory category;
    private boolean isHidden = false;
    @ColumnInfo(typeAffinity = ColumnInfo.BLOB)
    private byte[] sketchData;
    @TypeConverters(AttachmentConverter.class)
    private List<Attachment> attachments;
    private String taskColor; //default light grey color #E1E1E1
    // @TypeConverters(DateConverter.class)
    private Date creationDate;
```

How I used it in the ListFragment:

```
if (isSelectedAppointment) {
    taskFactory = new TaskAppointmentFactory();
    viewModel.insertAppointment((TaskAppointment) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, subtasks, attachments, sketchData, taskColor));
} else if (isSelectedChecklist) {
    taskFactory = new TaskChecklistFactory();
    viewModel.insertChecklist((TaskChecklist) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, subtasks, attachments, sketchData, taskColor));
}
```

1.3.3 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.
Abstract class ASubtask and its abstract methods which will be implemented in inherited concrete classes:

```
public abstract class ASubtask implements Parcelable {

    @PrimaryKey(autoGenerate = true)
    private int id;
    protected EStatus state;
    private String name;
```

```
abstract public void addSubtask(ASubtask subtask);
abstract public void removeSubtask(ASubtask subtask);
abstract public void setState(EStatus state);
abstract public void removeAllSubtasks();
abstract public void setSubtasks(List<ASubtask> subtasks);
abstract public List<ASubtask> getSubtasks();
```

Composite class containing the list of abstract subtasks:

```
public class SubtaskList extends ASubtask {

    private List<ASubtask> subtasks;

    public SubtaskList(String name) {
```

And SubtaskItem which realises the “leaf”

```
public class SubtaskItem extends ASubtask {

    public SubtaskItem(String name) { super(name); }
```

1.3.4 Iterator pattern

Iterator is a behavioural 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 subtasks tree structure.
Abstract iterator and list classes:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
    T currentItem();
    void reset();
}
```

```
public interface IterableList<E> {
    Iterator<E> iterator();
}
```

Here is the implementation of iterator and list:

```
public class SubtaskIterator implements Iterator<ASubtask> {
    private ASubtask[] subtasks;
    private int position;
```

```
public class SubtaskIterableList implements IterableList<ASubtask> {
    private ASubtask[] subtasks;

    public SubtaskIterableList(ASubtask[] subtasks)
```

And here is how the iterator pattern works inside SubtaskList (for composite pattern).

```
@Override
public void setState(EStatus state) {
    /**
     * When state changed in a parent subtask, iterate through children subtasks and set new state according to the parent
     */
    int subtasksCount = subtasks.size();
    IterableList<ASubtask> subtaskList = new SubtaskIterableList(subtasks.toArray(new ASubtask[subtasksCount]));
    Iterator<ASubtask> iterator = subtaskList.iterator();
    while(iterator.hasNext()){
        ASubtask nextSubtask = iterator.next();
        nextSubtask.setState(state);
    }
    this.state = state;
}
```

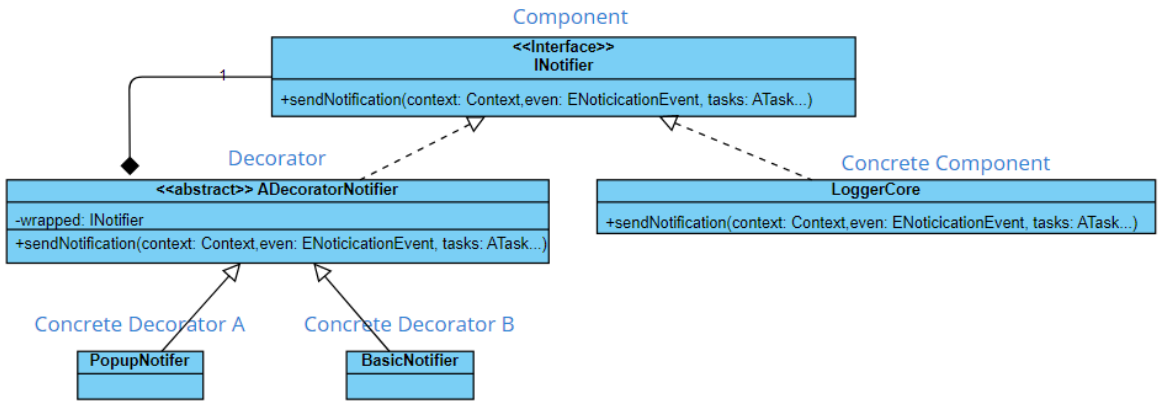
1.3.5 Decorator Pattern

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 behaviours. It is a flexible alternative to subclassing where whatever combination of behaviours can be specified without having to create a new class that comprises all behaviours for each combination.

We decided to use the decorator pattern for our notifications (member 3 responsibility). The users can dynamically select how they want to be notified about events, whether by a basic 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 basic notification as the decorators (optional wrappers) that contain the additional behaviours.

If we were to use subclassing to realise our notification system then we would have had to create a new class for each new combination of notification types, e.g. new class BasicAndPopupNotifier if we wanted popup and basic 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

Decorator Pattern




```java
public class LoggerCore implements INotifier {
    private static final String TAG = "LOGGER_CORE";

    @Override
    public void sendNotification(Context context, ENotificationEvent event, ATask... tasks) {
        for(ATask task : tasks){
            Log.d(event.name(), task.getTaskName());
        }
    }
```

```java
public abstract class ADecoratorNotifier implements INotifier {

    private INotifier wrapped;

    protected ADecoratorNotifier(INotifier wrapped) { this.wrapped = wrapped; }

    @Override
    public void sendNotification(Context context, ENotificationEvent event, ATask... tasks) {
        wrapped.sendNotification(context, event, tasks);
    }
```

```java
public class BasicNotifier extends ADecoratorNotifier {
    private static final String TAG = "BASIC_NOTIFIER";

    public BasicNotifier(INotifier wrapped) { super(wrapped); }

    @Override
    public void sendNotification(Context context, ENotificationEvent event, ATask... tasks) {
        super.sendNotification(context, event, tasks);

        for(ATask task : tasks){
            NotificationCompat.Builder notificationBuilder = new NotificationCompat.Builder(context,  channelId: "notifications")
                    .setSmallIcon(android.R.drawable.stat_notify_sync)
                    .setContentTitle(event.name())
                    .setContentText(task.getTaskName());

            Notification notification = notificationBuilder.build();

            NotificationManagerCompat notificationManagerCompat = NotificationManagerCompat.from(context);
            notificationManagerCompat.notify(task.getId(), notification);
        }
    }
```

In SettingFragment, we build a notifier depending on what was chosen in the settings.

```java
private INotifier buildNotifier(boolean popup, boolean basic) {
    INotifier notifier = new LoggerCore();
    if (popup) {
        notifier = new PopupNotifier(notifier);
    }
    if (basic) {
        notifier = new BasicNotifier(notifier);
    }
    return notifier;
}
```

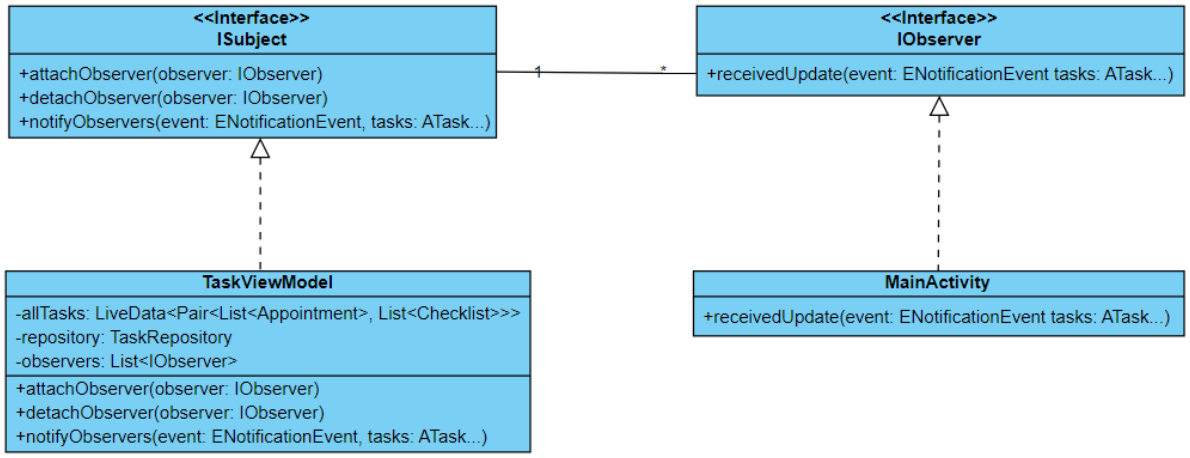
1.3.6 Observer Pattern

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 notifications (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 a notification must be sent out for upcoming appointments. The observable object in our case is the TaskViewModel and the observer is the MainActivity.

Without the observer pattern the MainActivity would have to constantly inquire about the state of the TaskViewModel/Database (was a task added, updated or deleted). This inquiry is really resource intensive which is why we decided to use the observer pattern here. Instead of constantly asking if the database has changed, the TaskViewModel itself notifies all observers when a task is added, updated or deleted and the observers can then do whatever they want to do with that information. In our case the MainActivity sends out notifications depending on the settings that were chosen by the user.

Observer Pattern



In MainActivity, we attach itself as an observer to TaskViewModel and implement receivedUpdate.

```
taskViewModel.attachObserver(this);
```

```
@Override
public void receivedUpdate(ENotificationEvent event, ATask... tasks) throws DeadlinePassedException {
    if (event == ENotificationEvent.CREATE) {
        eventNotifierViewModel.getOnCreateNotifier().sendNotification( context: this, event, tasks);
        for (ATask task : tasks) {
            if (task.getCategory().equals(ECategory.APPOINTMENT)) {
                setAlarm((TaskAppointment) task);
            }
            Log.d(TAG,  msg: "received onCreate update from taskViewModel: " + event.name() + " " + task.getTaskName());
        }
    }
    if (event == ENotificationEvent.UPDATE) {
        eventNotifierViewModel.getOnUpdateNotifier().sendNotification( context: this, event, tasks);
        for (ATask task : tasks) {
            Log.d(TAG,  msg: "received onUpdate update from taskViewModel: " + event.name() + " " + task.getTaskName());
        }
    }
    if (event == ENotificationEvent.DELETE) {
        eventNotifierViewModel.getOnDeleteNotifier().sendNotification( context: this, event, tasks);
        for (ATask task : tasks) {
            if (task.getCategory().equals(ECategory.APPOINTMENT)) {
                cancelAlarm((TaskAppointment) task);
            }
            Log.d(TAG,  msg: "received onDelete update from taskViewModel: " + event.name() + " " + task.getTaskName());
        }
    }
}
```

In TaskViewModel, we have a list of observers and we notify them when something has changed.

```
public void insertAppointment(TaskAppointment task){
    long insertedTaskId = repository.insertTaskAppointment(task);
    task.setId((int)insertedTaskId);
    notifyObservers(ENotificationEvent.CREATE, task);
}

public void updateAppointment(TaskAppointment task) {
    repository.updateTaskAppointment(task);
    notifyObservers(ENotificationEvent.UPDATE, task);
}
```

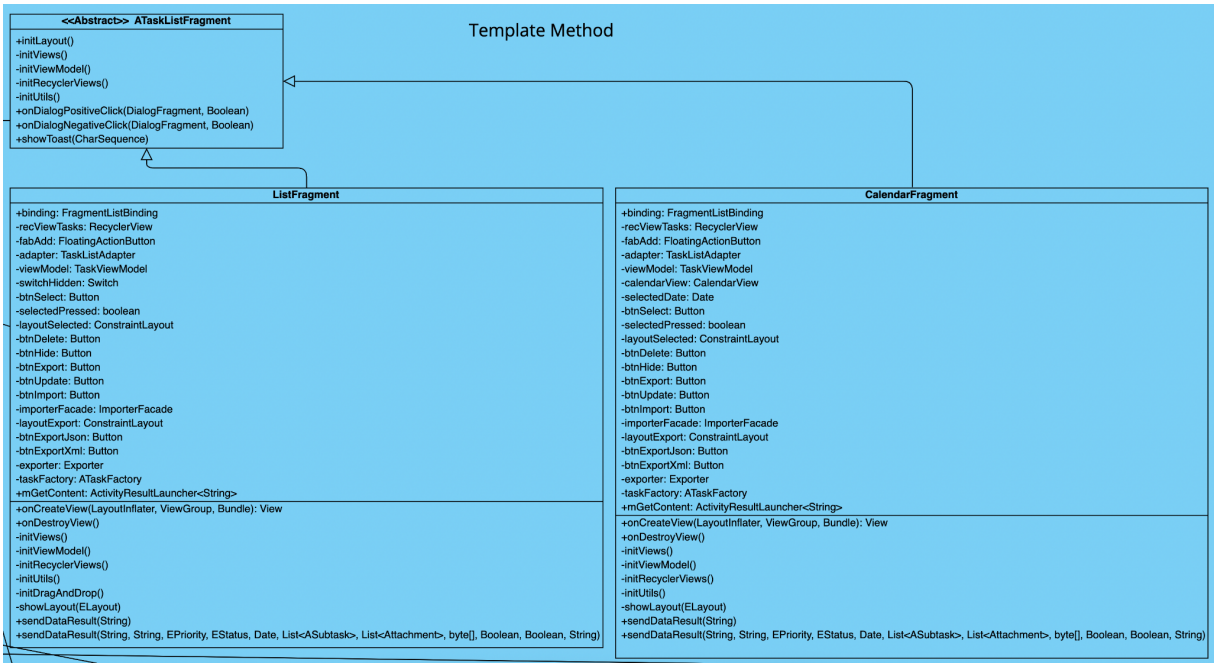
1.3.7 Template Pattern

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.

UML diagram was modified after the design phase to be compliant with the real fragments structure, relevant variables and methods.



Here is the implementation of the abstract class necessary for this pattern and both fragments that extends this abstract class:

```
public abstract class ATaskListFragment extends Fragment implements AddTaskFragment.AddTaskDialogListener,
        AddTaskFragment.SendDataFromAddDialog, PropertyToBeUpdated.SelectPropertyToUpdateDialogListener,
        PropertyToBeUpdated.SendDataFromSelectPropertyUpdateDialog {
    /**
     * Layout initialization function with steps that are overwritten on specific fragments
     */
    public void initLayout() {
        initViews();
        initViewModel();
        initRecyclerViews();
        initUtils();
    }

    protected abstract void initViews();
    protected abstract void initViewModel();
    protected abstract void initRecyclerViews();
    protected abstract void initUtils();
```

```
public class ListFragment extends ATaskListFragment {
```

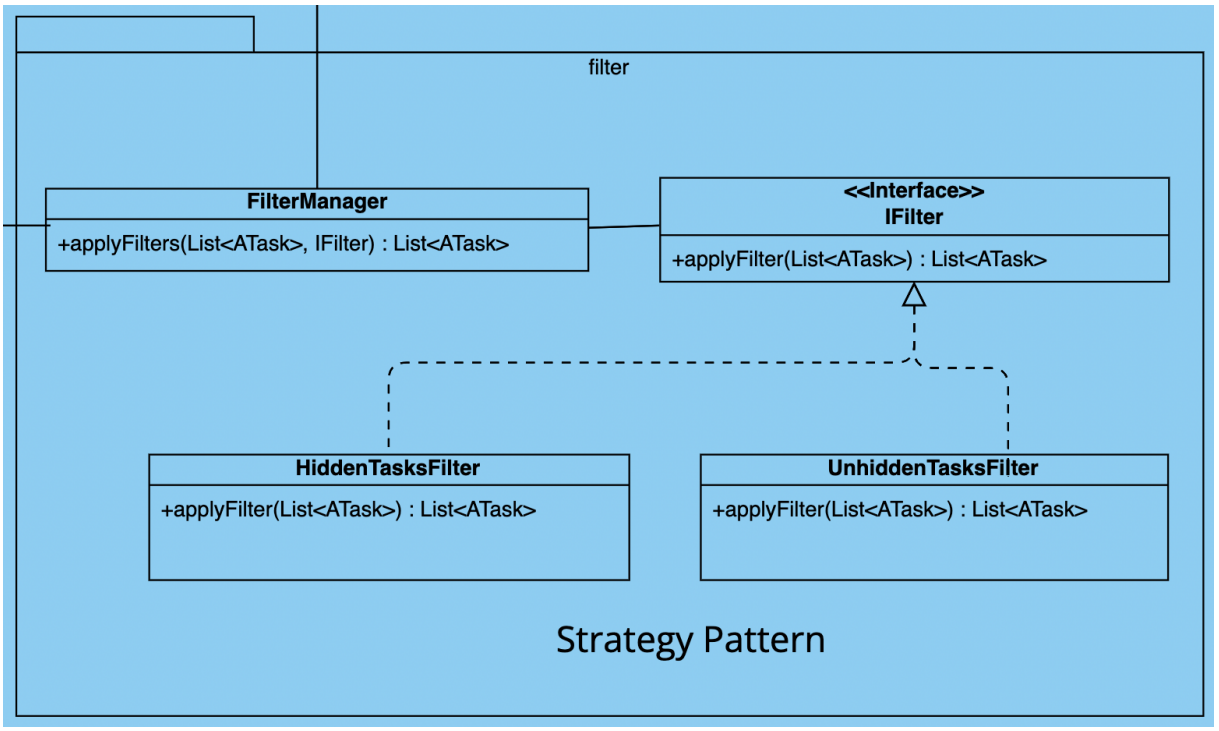
```
public class CalendarFragment extends ATaskListFragment {
```

1.3.8 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.

Based on the design phase feedback, inheritance instead of realisation was changed for the classes that implement interface.

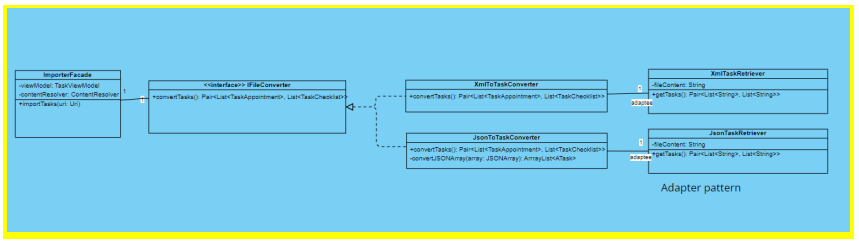


Here is the Filter Manager that holds a single function that includes task list and necessary filter as parameters.

```java
public class FilterManager {

    /**
     Class has single function to apply the chosen filter to the task list.
     @param taskList list of ATask objects
     @param filter filter interface
     @return list of ATask with filter applied
     */

    public List<ATask> applyFilter(List<ATask> taskList, IFilter filter) {
        List<ATask> filteredList = new ArrayList<>(taskList);
        filteredList = filter.applyFilter(taskList);
        return filteredList;
    }
}
```

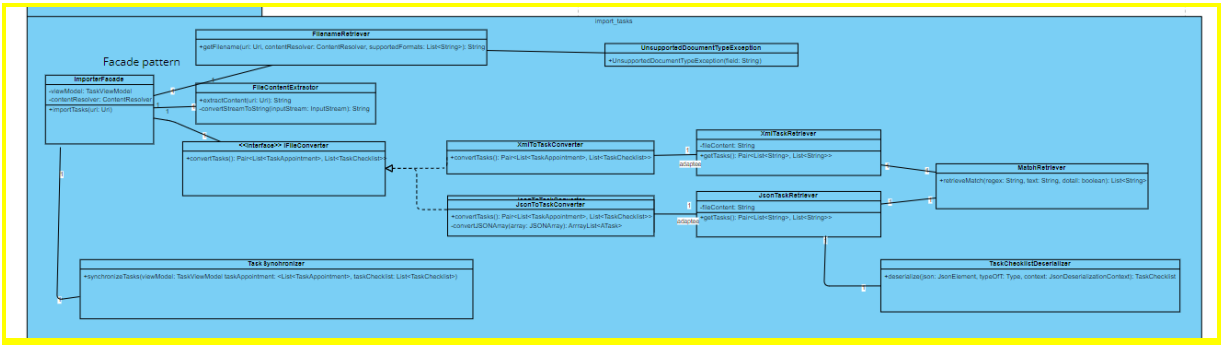
And as an example class for filtering hidden tasks that implements IFilter interface and could be used later in Filter Manager.

```java
public class HiddenTasksFilter implements IFilter {

    /**
     * Single function to apply filter to the task list
     * @param taskList list of ATask objects
     * @return list of hidden tasks
     */
    @Override
    public List<ATask> applyFilter(List<ATask> taskList) {
        List<ATask> hiddenTasks = new ArrayList<>();
        for(ATask task : taskList) {
            if(task.isHidden()) {
                hiddenTasks.add(task);
            }
        }
        return hiddenTasks;
    }
}
```



1.3.9 Facade Pattern

1.3.10 AdapterPattern



2 Implementation

2.1 Overview of Main Modules and Components

2.2 Quality Requirements Coverage

QR1 Comment your code and provide a code documentation in an appropriate manner (e.g., use JavaDocs for Java):
All members committed the code wherever we felt was needed and we decide to use JavaDocs.

QR2 Your implementation must be in compliance with a style guide (e.g., use Google Java Style Guide3 for Java):
We use camel case for our class names, all interfaces start with “I”, enums with “E” and abstract class with “A”

QR3 Use common coding practices:
See 2.3

QR4 Use defensive programming:
See 2.5

QR5 Use key design principles:
See 2.4

QR6 Make sure that your implementation works properly by testing it thoroughly:
See

2.3 Coding Practices

Member 1

I followed the conventions that the team agreed upon in the beginning of this phase, such as naming abstract classes “A…”, interfaces “I…”, enums “E…”.

```
public abstract class ATaskFactory {
```

```
public interface ITaskList {
```

I added JavaDoc comments where I considered needed.
I used method names that are intention revealing.

```
protected void initViewModel() {
```

```
protected void initRecyclerViews() {
```

Member 2

From the beginning our team agreed on specific naming conventions, which I followed throughout the project (as described above).
Moreover, I tried to create self-explanatory methods' and variables' names to improve readability but I also used JavaDocs if some additional explanation was needed.

Member 3

- Interface names all start with "I"

```
public interface IObserver {
    void receivedUpdate(ENotificationEvent event, ATask... tasks) throws DeadlinePassedException
}
```

- Enum names all start with "E"

```
public enum ENotificationEvent {
    CREATE,
    DELETE,
    UPDATE,
    APPOINTMENT
}
```

- Abstract class names all start with "A"

```
public abstract class ADecoratorNotifier implements INotifier {
```

- Method names are all intention revealing

```
public void sendNotification(Context context, ENotificationEvent event, ATask... tasks) {
```

- Split code in smaller single responsibility methods to increase readability

```
initViews();
initListeners();
initCheckboxLayout();
setCurrentTheme();
```

- Added JavaDocs wherever I deemed necessary

```
/**
 * The users can dynamically select what notification settings they want by checking the checkboxes
 * and the settings are persistently saved in the room database.
 * Every time a change occurs, the insert method of EventNotifierViewModel is called. Insert insinuates that
 * a new entry is added into the database but in actuality the entry is updated because a entry with
 * the same primary key is replaced when trying to insert.
 *
 * @author Jin-Jin Lee
 */
public class SettingsFragment extends Fragment {
```

Member 4

Abstract class starts with “A”, interface with “I”. Comments and docs are added to simplify future development and clarity purposes.

```
public abstract class ATaskListFragment
```

```
public interface IFilter {
```

Member 5

Interface starts with “I”, I added “Enum” to the class names, for each class, I used debugger and used TAG to specify the class where the debugger is called.

2.4 Key Design Principles

State where and to what extent you have used the key design principles. ***Each team member*** *must discuss and show examples from the code* ***separately for each responsibility****.*

Member 1

I followed the principle “Code to an interface, not to an implementation” when using the Factory pattern method for example, as there I was working with the higher level component, the abstract ATask, instead of working with concrete components such as TaskAppointment or TaskChecklist.

```
if (isSelectedAppointment) {
    taskFactory = new TaskAppointmentFactory();
    viewModel.insertAppointment((TaskAppointment) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, subtasks, attachments, sketchData, taskColor));
} else if (isSelectedChecklist) {
    taskFactory = new TaskChecklistFactory();
    viewModel.insertChecklist((TaskChecklist) taskFactory.getNewTask(taskName, taskDescription, priorityEnum, statusEnum, deadline, subtasks, attachments, sketchData, taskColor));
}
```

I followed the Single Responsibility Principle through decomposing the responsibilities of the classes as much as possible, while still following the Android guidelines as well as the MVVM architecture.

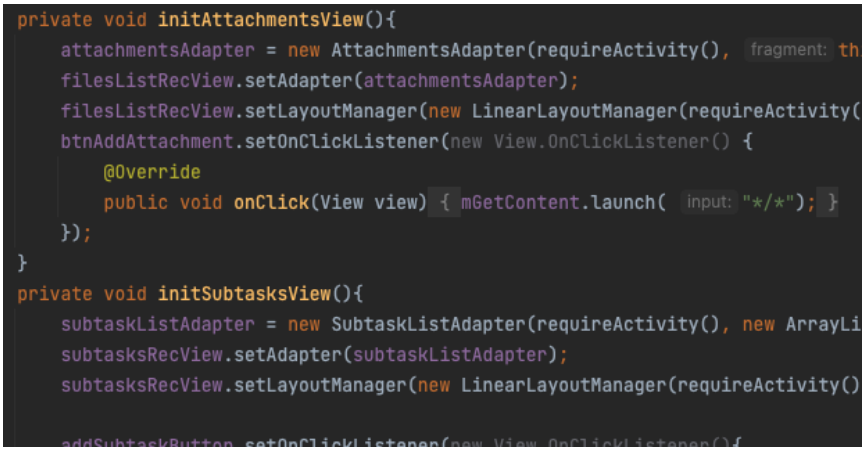
```
public void initLayout() {
    initViews();
    initViewModel();
    initRecyclerViews();
    initUtils();
}
```

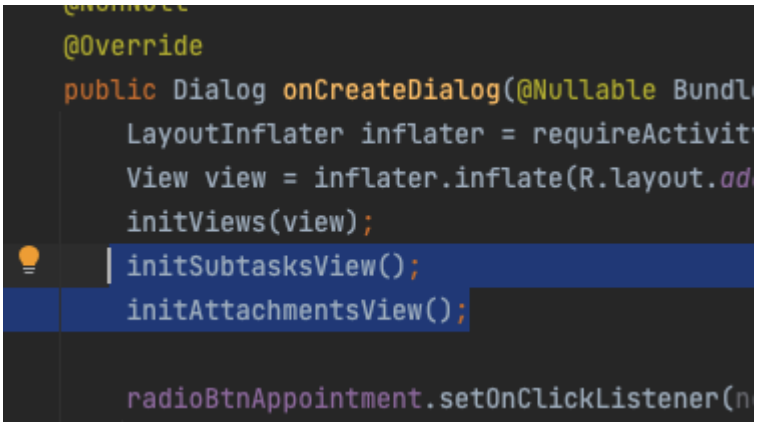
I followed the encapsulation principle by making everything that is not supposed to be visible private.

I followed the Open-Close Principle through the usage of the Factory Pattern: if we'd want to extend the app by adding other types of tasks, this wouldn't break the existing code.

Member 2

I followed the Open/Closed Principle by implementing the Composite pattern for the subtasks, where we can introduce new subtask types without breaking the existing code.
Also I used the Liskov Substitution Principle. While overriding methods I did not break their expected behaviour.
I followed Encapsulation where I provide public access only to the required functionality and others I leave private.
I followed the single responsibility principle by splitting complex code into smaller methods with only one responsibility.





Member 3

Single Responsibility Principle: I split my code into smaller single responsibility methods to increase readability

```
public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    binding = FragmentSettingsBinding.inflate(inflater, container,  attachToParent: false);
    eventNotifierViewModel = new ViewModelProvider(getActivity()).get(EventNotifierViewModel.class);

    initViews();
    initCheckboxListeners();
    initCheckboxLayout();

    return binding.getRoot();
}
```

Open/Closed Principle: The decorator pattern for the notifications follows the Open/Closed Principle because, we can easily extend our functionality with a new notification type without changing the core structure of our code.

```
private INotifier buildNotifier(boolean popup, boolean basic) {
    INotifier notifier = new LoggerCore();
    if (popup) {
        notifier = new PopupNotifier(notifier);
    }
    if (basic) {
        notifier = new BasicNotifier(notifier);
    }
    return notifier;
}
```

Don’t Repeat Yourself:
I extracted repeating code into a method instead of code duplication, e.g.

```
private INotifier buildNotifier(boolean popup, boolean basic) {
    INotifier notifier = new LoggerCore();
    if (popup) {
        notifier = new PopupNotifier(notifier);
    }
    if (basic) {
        notifier = new BasicNotifier(notifier);
    }
    return notifier;
}
```

Member 4

Open/Closed Principle: Strategy pattern and filter implementation is done to abide by that principle and could be easily extended in the future.

Encapsulation: private variables are provided as also methods that should not be visible, public methods are there to access or modify some elements. ATaskListFragment as an example:

```
public abstract class ATaskListFragment extends Fragment implements AddTaskFragment.AddTaskDialogListener,
        AddTaskFragment.SendDataFromAddDialog, PropertyToBeUpdated.SelectPropertyToUpdateDialogListener,
        PropertyToBeUpdated.SendDataFromSelectPropertyUpdateDialog {
    /**
     * Layout initialization function with steps that are overwritten on specific fragments
     */
    public void initLayout() {
        initViews();
        initViewModel();
        initRecyclerViews();
        initUtils();
    }

    protected abstract void initViews();
    protected abstract void initViewModel();
    protected abstract void initRecyclerViews();
    protected abstract void initUtils();
```

Single Responsibility Principle: making smaller methods that are used later and are easily understood by other developers and could be conveniently modified. Drag&Drop method on ListFragment as also initialising views, view models etc.

Member 5

I followed single responsibility principle and made each class only implement one functionality.

2.5 Defensive Programming

Member 1

For my responsibility, I used defensive programming in the case when the user wants to add a new task. There, when the user clicks on Add, if none of the task types is selected, the dialog doesn't get dismissed and a red validation text appears to inform the user that it must select a task type. This way, a task only gets forwarded to the database if it is created correctly. I considered it not to be necessary to add more validations to the task title or the task description for example, since I wanted to leave the user the possibility to add a task that doesn't have a title, nor a description or that has a title that contains numbers. These can be further updated if the user wants to.

```
if (!isSelectedAppointment && !isSelectedChecklist) {
    taskTypeValidation.setVisibility(View.VISIBLE);
    wantToCloseDialog = false;
    return;
} else {
    taskTypeValidation.setVisibility(View.GONE);
    wantToCloseDialog = true;
}

listener.onDialogPositiveClick( dialogFragment: AddTaskFragment.this, wantToCloseDialog);
```

I added an exception SingletonDbDoubleInitException, that gets thrown if for any reason the database gets created a second time.

Member 2

I used defensive programming for the case when a user tries to open the attached file, which does not exist anymore. In this situation, an OpenAttachmentException will be thrown and handled in a way that the user will be notified in a suitable manner about the issue. In this way I prevent the app from crashing in an unexpected situation.

```
holder.openFileBtn.setOnClickListener(view -> {
    Log.d(TAG, msg: "Open attachment button clicked");
    try {
        listener.openFile(this.attachments.get(position).getFilePath());
    } catch (OpenAttachmentException e) {
        holder.messageText.setText("File doesn't exist");
        holder.fileParentCard.setCardBackgroundColor(Color.GRAY);
        Log.d(TAG, msg: "Cannot open attachment: " + e.getMessage());
    }
});
```

I also used exceptions throughout the project. For example in SubtaskConverter it can be that invalid json was saved and it cannot be converted to a list of subtasks, so I just return an empty list and use logging to notify the developer about an occurred error.

```
} catch (JSONException e){
    Log.d(TAG, msg: "Error on deserializing a list of ASubtask: " + e.getMessage());
}
```

Member 3

I used defensive programming for the case when a user sets a deadline for an appointment that has already passed. In this situation a DeadlinePassedException is thrown and no alarm for that particular appointment is set.

```
if (currentTime.after(deadline)) {
    throw new DeadlinePassedException(currentTime, deadline);
}
```

I also used logging throughout the project and checked if values are null before continuing with further steps.

```
if (event == ENotificationEvent.DELETE) {
    eventNotifierViewModel.getOnDeleteNotifier().sendNotification( context: this, event, tasks);
    for (ATask task : tasks) {
        if (task.getCategory().equals(ECategory.APPOINTMENT)) {
            cancelAlarm((TaskAppointment) task);
        }
        Log.d(TAG,  msg: "received onDelete update from taskViewModel: " + event.name() + " " + task.getTaskName());
    }
}
```

```
if(intent != null){
    String notifierString = intent.getStringExtra(NOTIFIER_KEY);
    INotifier notifier = notifierTypeConverter.toINotifier(notifierString);

    ATask appointment = intent.getParcelableExtra(APPOINTMENT_KEY);
    notifier.sendNotification(context, ENotificationEvent.APPOINTMENT, appointment);
}
```

Member 4

I used defensive programming for the case when FilterManager could receive null task list and when saving the initial app theme persistently in the app.
First case was achieved by providing @NonNull before task list value in the applyFilter function.
Second case was achieved by providing default value as light app theme in the switch case when the user opens app for the first time.

```
String sTheme = sharedPreferences.getString( s: "theme",  s1: ""); // "theme" is key and second "" is default value

switch(sTheme){
    case "light":
        Log.d(TAG,  msg: "Current app theme: light");
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO);
        break;
    case "dark":
        Log.d(TAG,  msg: "Current app theme: dark");
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES);
        break;
    default:
        Log.d(TAG,  msg: "Default app theme: light");
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO);
        break;
}
```

Member 5

I made sure I catch each exception separately and also used assert to make sure that I do not get any unexpected arguments.

2.6 Testing

*Describe how you tested the implemented functionalities. **Each team member** must discuss and show examples from the code **separately for each responsibility**.*

Member 1

I took every responsibility that I had to implement and made a different class of tests for each.
For the addition of a new task, I tested whether it gets inserted into the database or not (AddTaskTest).
For the deletion of a task, I tested whether it gets deleted from the database (DeleteTaskTest).
For the update feature of a task, I tested whether the properties get updated correctly in the database (UpdateTaskTest) but also if the common update property feature works (UpdateCommonPropertyTest).
I also tested whether the selection works properly (SelectDeselectTest).
I tested the parcelable implementations of both task types (ParcelableTaskAppointmentTest, ParcelableTaskChecklistTest).
I tested the functionality of the layout when pressing the Select button on the listFragment (PressSelectButtonTest).
I tested whether the validation of task type when adding a new task works. (UnselectedTaskTypeValidationTest)

Member 2

For every class that I created I wrote a test class where I tested public methods and their expected output.
To test if the subtasks are correctly converted to JSON in SubtasksConverter, I created SubtaskConverterTest class and split the test methods into different cases such as check the output when 1) invalid json is given; 2) valid json and with multi level subtasks; 3) and when a list of multi level subtasks is given. For the last case I created a json string as the expected result which I save to the test class variable on setUp.
I also tested the Attachment class if it is working correctly with Parcel.
And in another SubtaskTest class a special case for the SubtaskList, where setting the state property in parent, should also be set to all children subtasks.

Member 3

I created a test to see if an DeadlinePassedException is thrown when the user sets a deadline that is already over. I also created a test to see if an alarm for an upcoming appointment is successfully set if the deadline is valid.

I tested if the INotifierTypeConvert correctly converts from INotifer to String and the other way around.

Member 4

I created tests for updating hidden task value, test using mocking for the abstract class of task list fragment and additionally tests for FilterManager and filtering through tasks.

Member 5

3 Code Metrics

We analysed the code metrics using the default lint tool provided by the Android Studio. And we also used an additional plugin - “Statistic” from IntelliJ - to better analyse 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: 10
Number of classes: 114

Lines of code

- in Java classes: 6678
- In XML: 1907
- comments in classes: 544

Current bugs

- open file button on the attachment card has not been implemented yet to open a file (this function was not required from the task specification for Member 2 responsibility) but it still has a function (read Member 2 Contribution for more details)
- drag&drop does not change task list order persistently (is not required by the specification)

4 Team Contribution

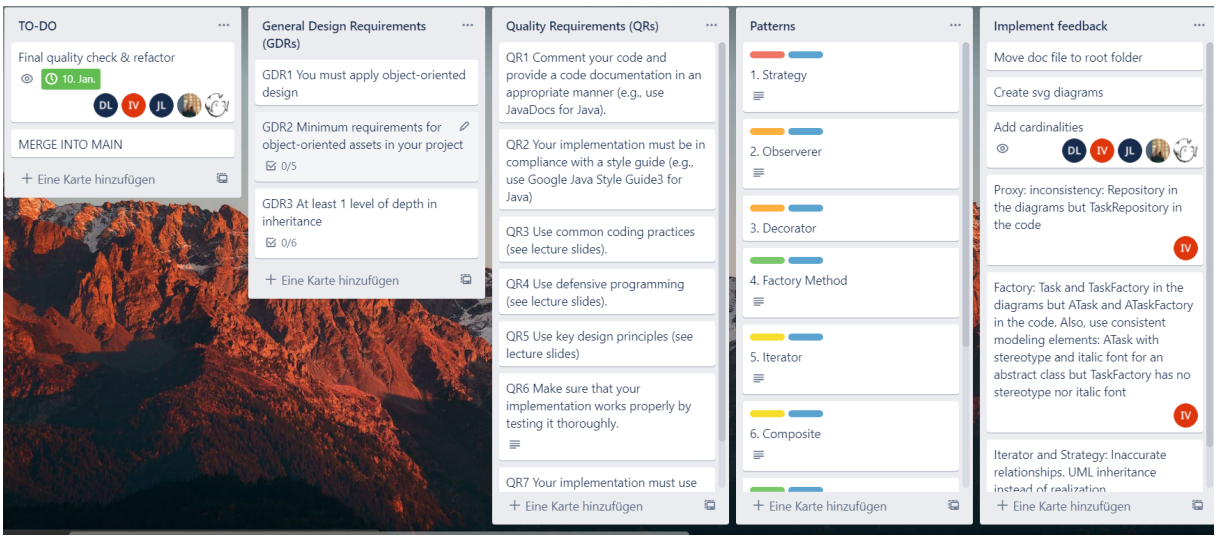
4.1 Project Tasks and Schedule

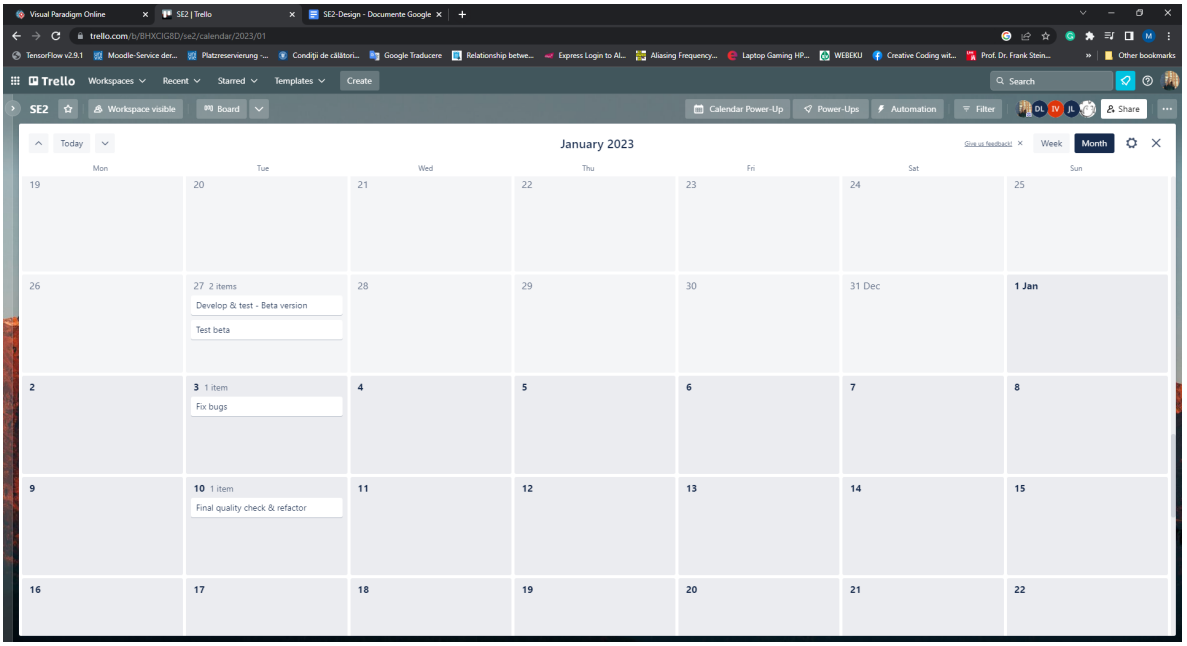
In order to have a good overview over the main requirements, details and deadlines of this project, we have used Trello from the very beginning of our group work.

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.





4.2 Distribution of Work and Efforts

Each member managed to successfully implement all their requirements on time. Whenever someone was having difficulties or wasn't sure which way to approach, we would all discuss it and try to solve the problem or give advice.

Contribution of member 1: (Factory Pattern, Proxy Pattern)
As most of the project was based on the implementation of the member 1 responsibilities, I tried to finish my code as soon as possible so that my colleagues could start implementing theirs. I put a lot of thought into the design phase on how to plan my code so that I wouldn't be having too many bugs to fix or problems during the final phase. This approach has helped me and so I was able to finish the requirements in the beginning, keeping the rest of time for bug fixes and changes needed due to my colleague's implementations.

Contribution of member 2: (Composite Pattern, Iterator Pattern)
In the final phase I implemented Composite and Iterator Patterns. For both patterns I needed to apply some changes compared with the design phase (which I described in section 1.2). I implemented nested subtasks for TaskChecklist with an appropriate deletion strategy. Then for both task types I implemented an option to choose files from the system and attach them to the task. Additionally, I implemented a Sketch option. It can be created for any task type and attached as an image. Finally, I wrote unit tests for the above functionality.

Contribution of member 3: (Decorator Pattern, Observer Pattern)
For the final phase I implemented the decorator pattern as well as the observer pattern. The decorator pattern stayed the same as in the design phase (except names) and the observer pattern has undergone major changes after I realised that it didn't make much sense the way it was in our initial design (see 1.2). I had to change the core structure of our app to accommodate fragments for list, calendar and

settings as opposed to just the one MainActivity in our design phase. I also had to make slight adjustments and additions to my colleagues code like adding a timepicker for appointments. All settings are saved in the room database persistently. At the end I wrote some tests to test my code.

Contribution of member 4(Template Pattern, Strategy Pattern):
In the design phase it was already established how the code should be structured and how the final product should look like. In the final phase, first of all for my part I started with Template Pattern and realised abstract class for fragments that are used on 2 tabs in our app: main task list and calendar view. For that I restructured already existing code only a little bit. Strategy pattern for the hidden/unhidden tasks filters was implemented exactly as it was designed in the previous phase. Apart from that smaller things were done: switch between hidden and unhidden tasks on the main screen with the help of the filter, color picker and change of colors for tasks with the help of external library, simple drag&drop on main screen with a few fixes in comparison to design phase, and switching between app themes with saving the state of the theme persistently.

Contribution of member 5: (Adapter Pattern, Facade Pattern)

4.3 How-To Documentation

To launch the application, please download the app-debug.apk provided in the implementation/app/build/outputs/apk/debug folder. Then drag and drop the apk into the AVD to install it.

Here we have a step-by-step tutorial on how to test all responsibilities that we have implemented:

Member 1 Responsibility - Testing

1. Add (Create) new task: to create a new task, click on the blue FloatingActionButton in the right bottom corner. Insert the data and select a type of Task (Appointment or Checklist). Once ready, click on “Add” to add the new task. The newly created task should appear in the list.
2. To view or update the task, click on the desired task. Once ready, click on the “Update Task” button at the bottom. Or click on “Cancel” to restore the changes.
3. To initiate the “Select Mode”, press on the “Select” Button. To select one or multiple tasks, press on the radio buttons at the left of the tasks.
4. To update a common property of one/multiple selected tasks, while in the Select Mode, select one/multiple tasks and click on the “Update” button from the list at the bottom. This opens a dialog. Press “Update” to update the common property. Or press “Cancel” to go back.
5. To delete one/multiple selected tasks, while in the Select Mode, press on the “Delete” button from the list at the bottom.

Member 2 Responsibility - Testing

- open an existing task with checklist icon or add new task and press “checklist” radio button
- under the “Task status” press “Add subtask” button
- you can add another level of subtasks by pressing the green “+” button
- to delete (either single subtask or a tree) press on the red “rubbish” button

To create a sketch (for any type of the task):

- press on “Create sketch” -> a window for drawing should appear
- draw and press on “Create” button if you want to save the sketch and on “Cancel” if you don’t
- if you pressed “Create” the sketch will appear in the task below as an image

To add a list of attachments (for any type of the task):

- press on “Attach file”
- choose any file from the system
- an attachment card will appear in the task view
- you can add other attachments by following the same steps
- to delete it press on red cross
- open file button (on the left from the red “cross”) won’t open the file (this is scheduled for the future implementation) but if you try to press on it after the attached file was deleted from the system, you will be informed about it (the card should change colour to grey and an appropriate message should appear

Member 3 Responsibility - Testing

Notifications on creation of a task

1. Press the “settings” tab at the right bottom corner and select the desired notification types by clicking on the checkboxes.
2. Go to the “list” tab at the left bottom corner and create a task
3. Notifications should appear according to the chosen settings.

Notifications on update of a task

1. Press the “settings” tab at the right bottom corner and select the desired notification types by clicking on the checkboxes.
2. Go to the “list” tab at the left bottom corner and click on a task to open it.
3. Change some properties e.g. task name
4. Notifications should appear according to the chosen settings.

Notifications on deletion of a task

1. Press the “settings” tab at the right bottom corner and select the desired notification types by clicking on the checkboxes.
2. Go to the “list” tab at the left bottom corner and delete a task.
3. Notifications should appear according to the chosen settings.

Notifications for upcoming appointments

1. press the “settings” tab at the right bottom corner and select the desired notification types by clicking on the checkboxes.
2. Create an appointment with a deadline that has not passed yet
3. Wait until the deadline is due and notifications should appear according to the chosen settings.

Member 4 Responsibility - Testing

1. Calendar View: create a few appointment tasks for different dates. After that step go to the Calendar tab on the bottom of the screen. There you will see the calendar, press on the date for which appointment tasks were created, you should see them under the calendar.
2. Drag&Drop: on the main screen with a task list, create a few tasks if the list is empty. Press and hold on the desired task after which you can move it through the list and place it on the desired spot.
3. Task Color: Press on the existing task on the main screen, on task info screen press on pick task color and choose desired color on the popup view. Press on update task, now background task color should be changed to the desired one.
4. Hiding/Unhiding tasks: On the task list with already created few tasks press on “Select” button on the right top of the screen, choose tasks you want to hide and press on the “Hide” button. After that you will be able to find hidden tasks if you press on the “Show hidden tasks” switch. On that view with the same steps you will be able to also unhide the tasks.
5. App theme: Press on the “Settings” tab on the bottom of the screen, there under “App Theme” press on “Dark Theme” switch to turn on dark mode for the app. Press on the switch again to turn it off, it will bring back light theme to the app.

Member 5 Responsibility - Testing