Software Engineering 2
FINAL REPORT

Team number:	0202

Team member 1	
Name:	Nikola Bicanic
Student ID:	11911340
E-mail address:	a11911340@unet.univie.ac.at

Team member 2	
Name:	Nikita Glebov
Student ID:	01468381
E-mail address:	nikegleb@yande.ru

Team member 3	
Name:	Dana Altman
Student ID:	01468758
E-mail address:	a01468758@unet.univie.ac.at

Team member 4	
Name:	Daryna Vandzhura
Student ID:	01409653
E-mail address:	a01409653@unet.univie.ac.at

Team member 5	
Name:	Sofiia Badera
Student ID:	11715248
E-mail address:	a11715248@unet.univie.ac.at

1 Final Design

1.1 Design Approach and Overview

The application design is using the Model-View-ViewModel (MVVM) software design pattern that is structured to separate program logic and user interface controls.

The *Model* layer is responsible for all business logic of the application and provides all the necessary data.

The *View* layer represents functionality to a user.

The *ViewModel* layer is an abstraction of the view exposing public properties and commands.

Implemented Patterns

1. Factory Pattern is used in the creation of the Subclasses for the Task class.

   And also this pattern is used in order to create the Notification class. As we have two notification types Email and Popup, sending notifications in these classes will be implemented in different ways.

2. Proxy Pattern is implemented with instance ProxyTask which will hold all of the created Tasks,control their creation, access, storing and various manipulations of the data.

3. Composite Pattern is used to create a system, which contains Tasks, that contains Subtasks, Attachment etc.

4. Strategy Pattern is suitable for deletion of parent and/or subtasks (e.g. how to handle the deletion of a task that contains subtasks).

5. Observer Pattern is used for notification sending related functionality. This pattern provides a solution for the case, when application user should be notified, when a task is created, updated, deleted and when an appointment is coming up.

6. Decorator Pattern is used to decorate Email and Pop-up Notifications according to the action.

7. Facade Pattern provides access to CalendarViewImpl and ListViewImpl classes. It provides a simplified interface to a complex system of sub-classes.

8. Template Method Pattern defines the steps for editing calendar and list views, the steps are overridden in subclasses for each view.

9. Adapter Pattern is used to work with and parse JSON and XML files.

10. Iterator Pattern is used to iterate the task list that is to be exported or imported from the database.

Application components for database

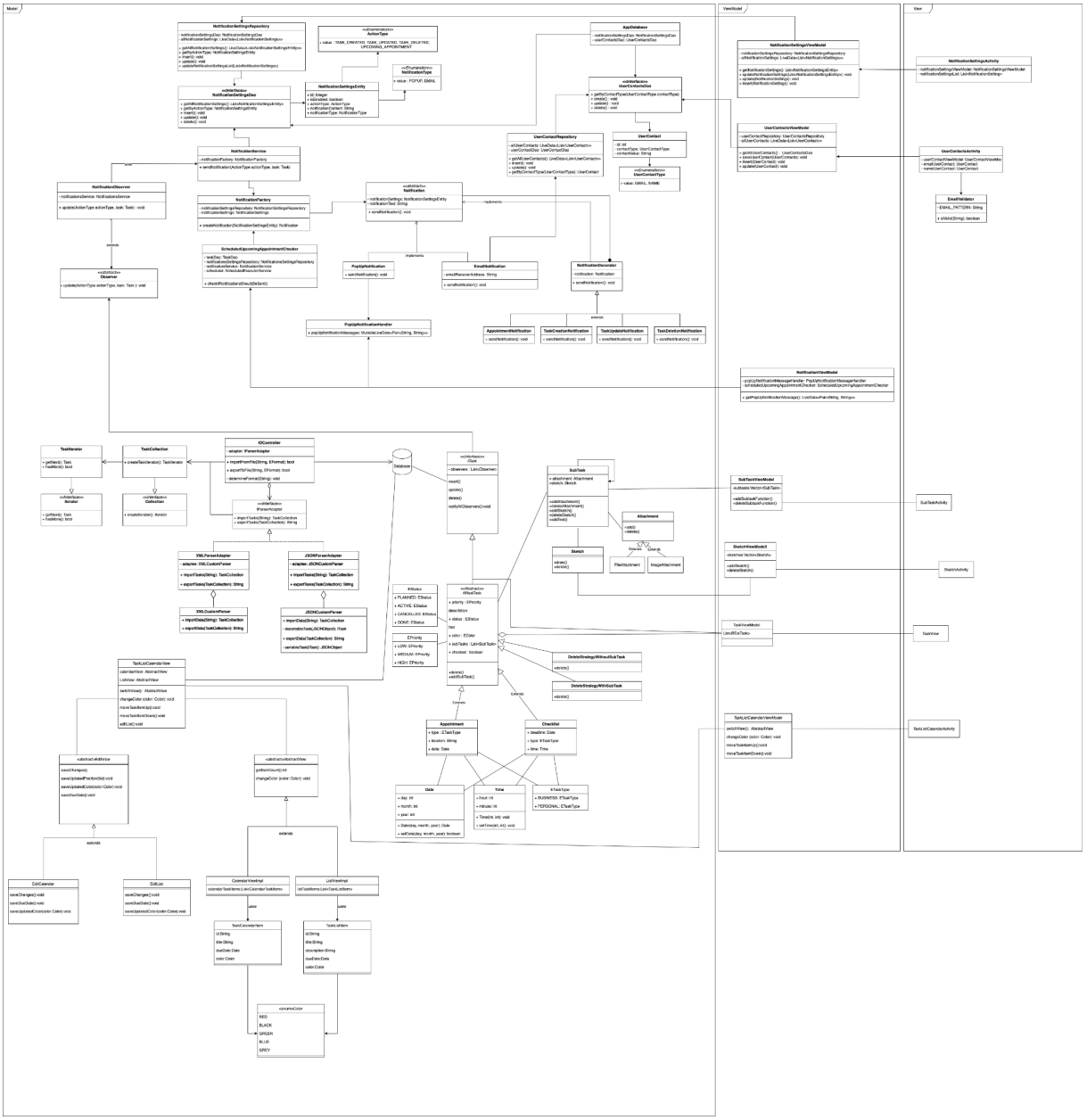
Since we decided to use the Room persistence library for SQLite, we created a database design architecture according to the guidelines from the site https://developer.android.com. Primary database design components are: database class, Data entity classes, Data Access Objects (DAOs) classes and repository classes. Database class represents the main access point to the database. Data entity classes represent tables in our database. DAOs provide CRUD methods for managing data in the database from our application. Repository classes are the entry point for ViewModel to send and receive persisted data. These classes also map objects stored in the database to prepare data representation in application UI.

1.1.1 Class Diagrams

Task management application is being developed by following Model-View-ViewModel design pattern.

To represent functionality to a user, our task management application has a *View* layer, which consists of Activity classes. These classes are responsible for defining UI components and ways of data representation. Classes from *View* layer send data to and receive data from classes of *ViewModel* layer. These classes are responsible for binding data from/for View. From the *ViewModel* layer data will be transferred to the *Model* layer.

The *Model* layer is responsible for all business logic of the application, including accessing and persisting data in the database, preparing and providing this data to *ViewModel* layer, and other functionalities which are not visible to the customer.



Database components:

2 Abstract class *AppDatabase* which extends *RoomDatabase* and defines all database configurations. We are using this class as the main entry point to our persisted data.

3 Entity classes: *Time, Date, Checklist, Appointment, Task, NotificationSettings, UserContact*.

4 DAO classes: *AppointmentDao*, *ChecklistDao*, *NotificationSettingsDao*, *UserContactDao*

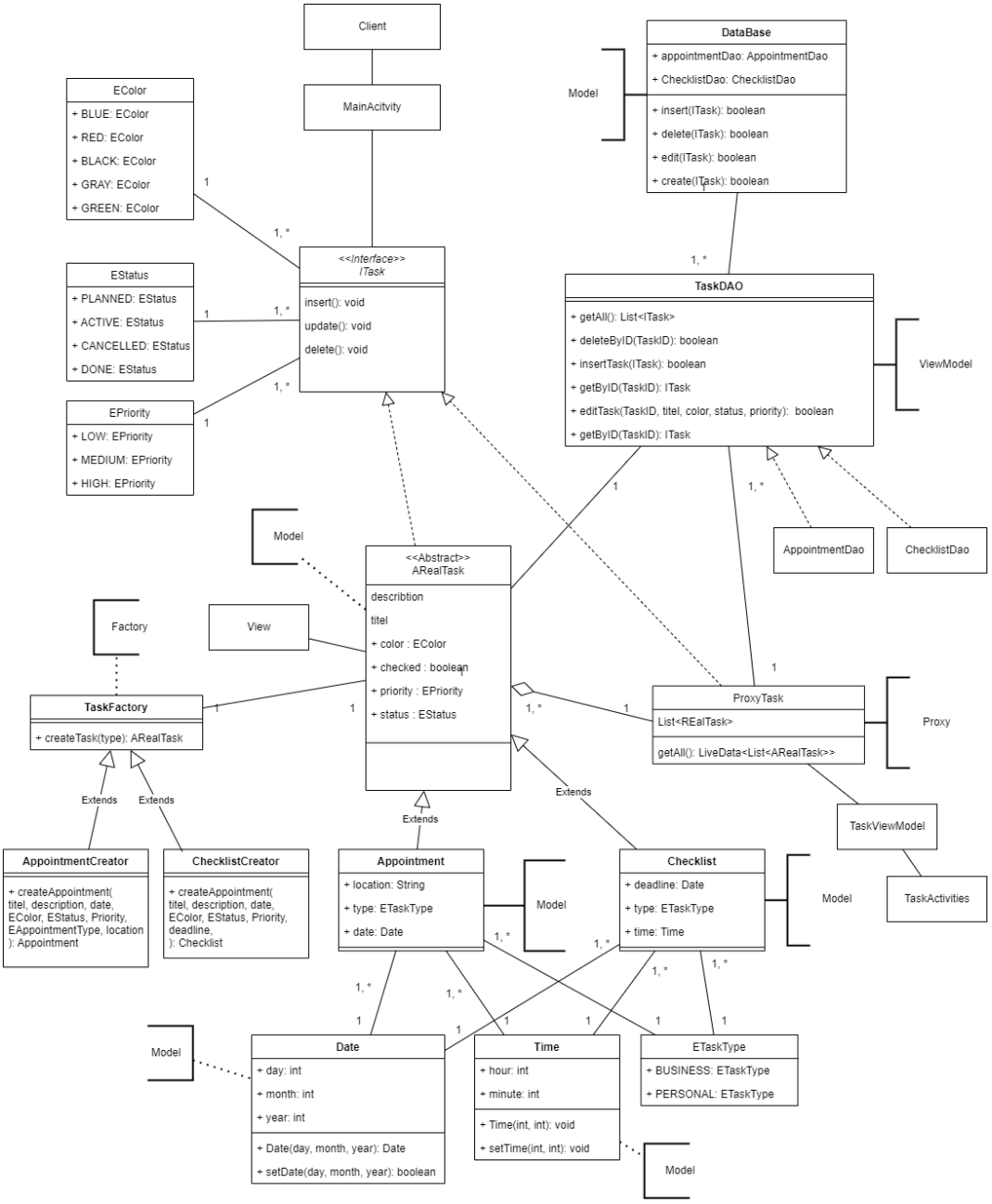
5 Repository classes: *NotificationSettingsRepository, UserContactRepository, TaskProxy.*

Member 1 responsibility

The main focus for this member will be the creation, editing, storing and deletion of Tasks. 2 Task Types will be available (Appointment and Checklist).

The idea to implement this is by using the Room Database provided by Android. There will be stored, updated and deleted all the Tasks created. In order to do this we will utilize a Dao which handles the actual insert, updating and deletion in the database.

An Interface ITask will be utilized to let the subclasses have the same set of abilities. This is needed to create a Proxy pattern (see point ..). A real Task and a Proxy Task will implement ITask and its methods. As for the real Task, this will serve as a super class for the concrete implementations of an Appointment Task and Checklist Task.

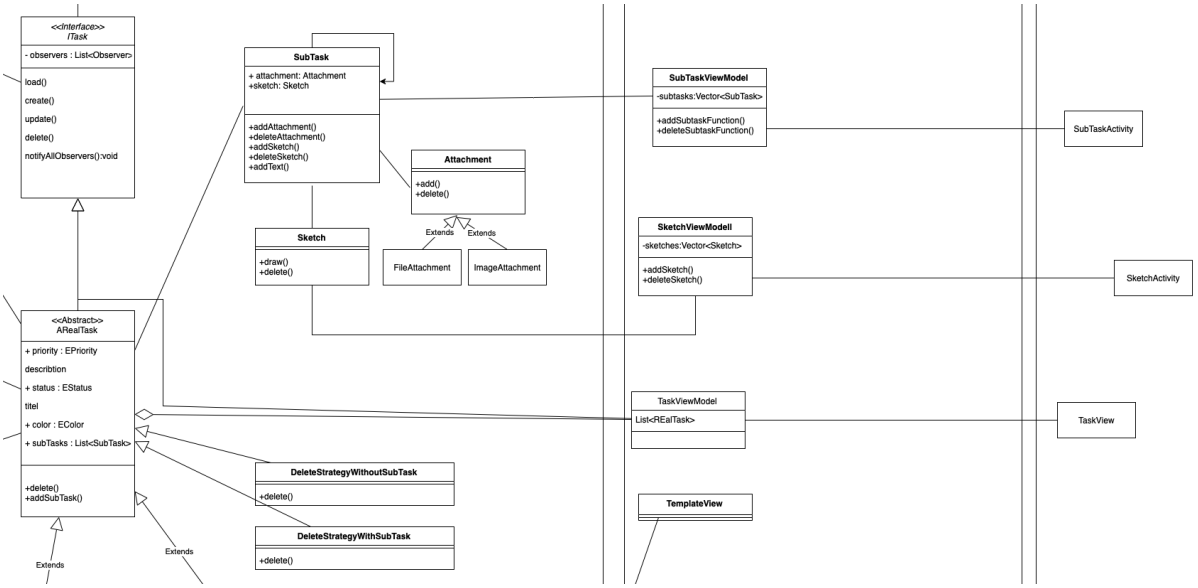


Member 2 responsibility

The Member 2 responsibility was to implement the suitable strategy for deletion of parent and/or subtasks, to ensure users can create a sub-task and to make hand written notes. To ensure that each task type supports at least two file format attachments.

View: SubTaskActivity informs the SubTaskViewModel about the adding and deleting subtasks. SketchActivity informs the SketchViewModel about the adding and deleting handwritten sketches.
ViewModel: SubTaskViewModel serves as a link between SubTaskActivity and Subtask, SketchViewModel between SketchAktivity and Sketch classes. Sketch class provides a possibility to add handwritten notes.

Model: class Attachment provides the interface for two types of attachment (file and image). FileAttachment and ImageAttachment are implementing the methods add() and delete() of the Attachment interface.
Class SubTask is responsible for the adding and deleting the Attachment,adding and deleting Sketches as well as adding text.



Member 3 responsibility

Task management application will support customizable notifications and users can dynamically select the actions to be notified: when a task is created, updated, deleted and an appointment is coming up.

All of classes, related to notifications functionality, can be divided into 3 groups of components for:

- creating and sending notifications;
- managing dynamically changeable notifications settings (e.g. content, notification type, etc);
- adding/editing user contact (e.g. email).

Design patterns: Observer, Factory, Decorator.

All of classes are structured by layers:

- View layer for notifications functionality consists of *NotificationSettingsActivity, UserContactActivity, EmailValidator* classes.
- ViewModel layer classes are: *NotificationSettingsViewModel, UserContactViewModel, NotificationViewModel.*
- Model layer components are: *NotificationSettingsEntity, UserContactEntity, NotificationSettingsDao, UserContactDao, NotificationSettingsRepository,*

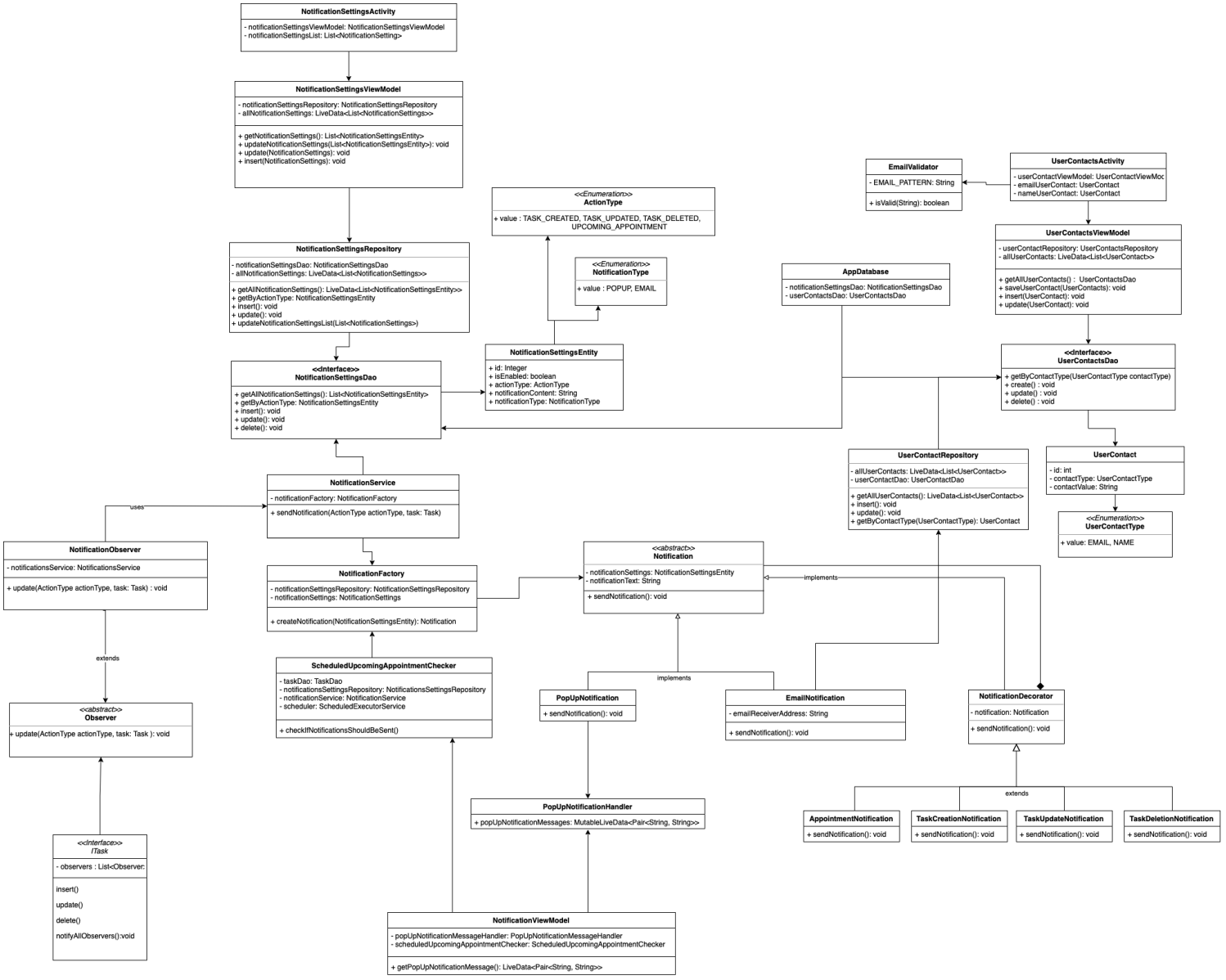
UserContactRepository, NotificationType, ActionType, NotificationService, NotificationFactory, ScheduledUpcomingAppointmentChecker, PopUpNotificationMessageHandler, Notification, PopUpNotification, EmailNotification, NotificationDecorator, AppointmentNotification, *TaskCreationNotification*, *TaskUpdateNotification*, *TaskDeletionNotification, Observer, NotificationObserver .*

- Exception Classes: *NotificationDisabledException, UnknownNotificationTypeException, UserContactNotFoundException*

The Model layer is providing data to ViewModel and receiving it back. The entry point in the Model layer, which is responsible for communicating with ViewModel, is Repository classes: *NotificationSettingsRepository* and *UserContactRepository.* These classes get and prepare data from the database for ViewModel and vice versa. Repository classes use CRUD operations from Dao classes to access and persist data in the database. For database objects representation we use Entity classes: *NotificationSettingsEntity* and *UserContactEntity.* For more structured data representation we added Enumeration classes: *NotificationType* and *ActionType.*

NotificationService class contains main orchestration logic for notifications functionalities. To get notified about changed task state, we use behavioural Observer pattern, which is implemented with the *Observer* abstract class and *NotificationObserver* class. This *NotificationObserver* class will be called, when the task state is changed, and in its turn will call *NotificationService*. For task creation we follow the creational Factory pattern by using *NotificationFactory* class. This class is used for creating different notifications. The *NotificationFactory* class can be called from *NotificationService* or from *ScheduledUpcomingAppointmentChecker* class. This ScheduledUpcomingAppointmentChecker class has a scheduled job, which checks if there is an upcoming appointment and a related notification should be sent.

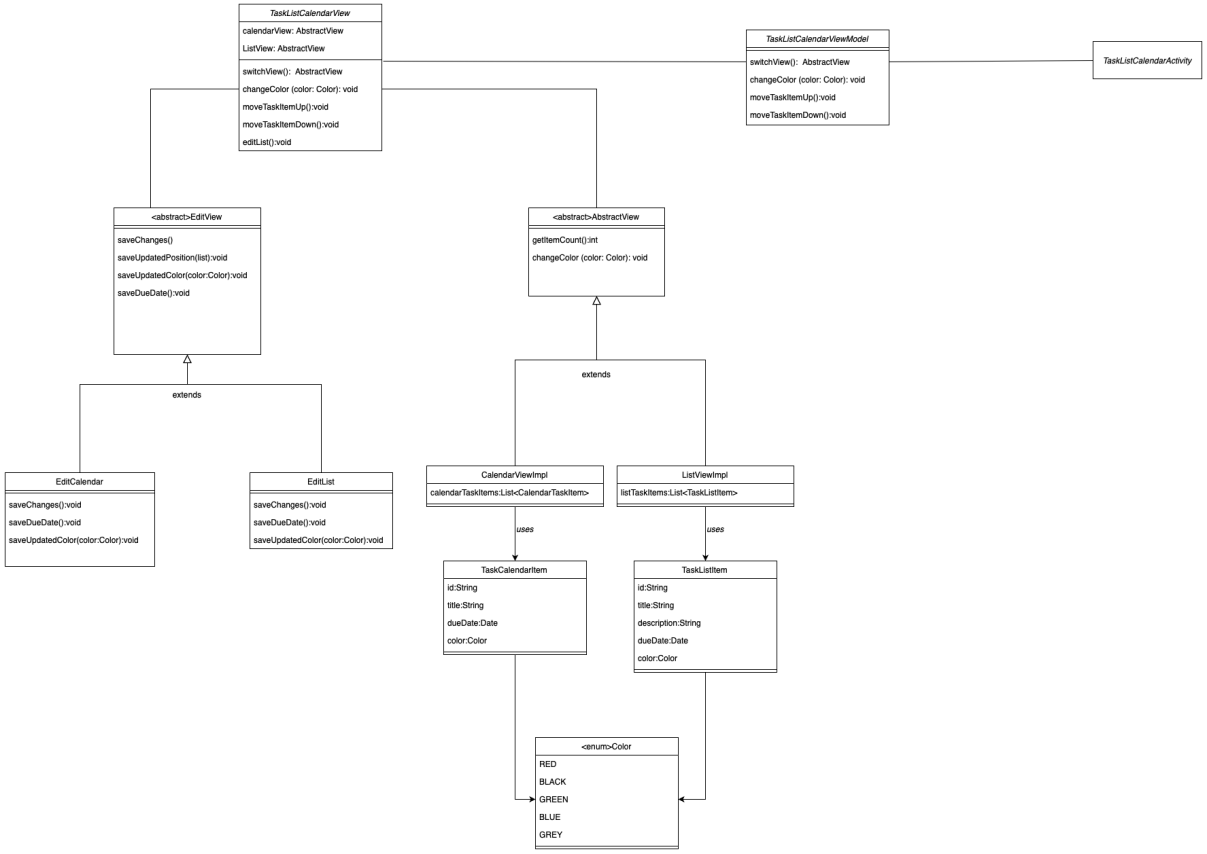
For providing different notification types we have a *Notification* interface, which is the parent of *PopUpNotification* and *EmailNotification* classes. To make customising notifications way more easier, we follow the Decorator pattern for sending notifications by different action types. Therefore we created the *NotificationDecorator* interface, which also extends *Notification* interface, and its children classes: *AppointmentNotification*, *TaskCreationNotification*, *TaskUpdateNotification*, *TaskDeletionNotification*.



Member 4 responsibility

The application supports customizable views, such as list view and calendar view. Also it should be possible for the user to reorder tasks in a list view and hide or show chosen tasks.
The MVVM approach is used here. It is designed in the following way:
View: TaskListCalendarActivity informs the ViewModel about the user's actions such as: change colour of element, move task up or down, switch views.
ViewModel: TaskListCalendarViewModel serves as a link between TaskListCalendarActivity and TaskListCalendarView.

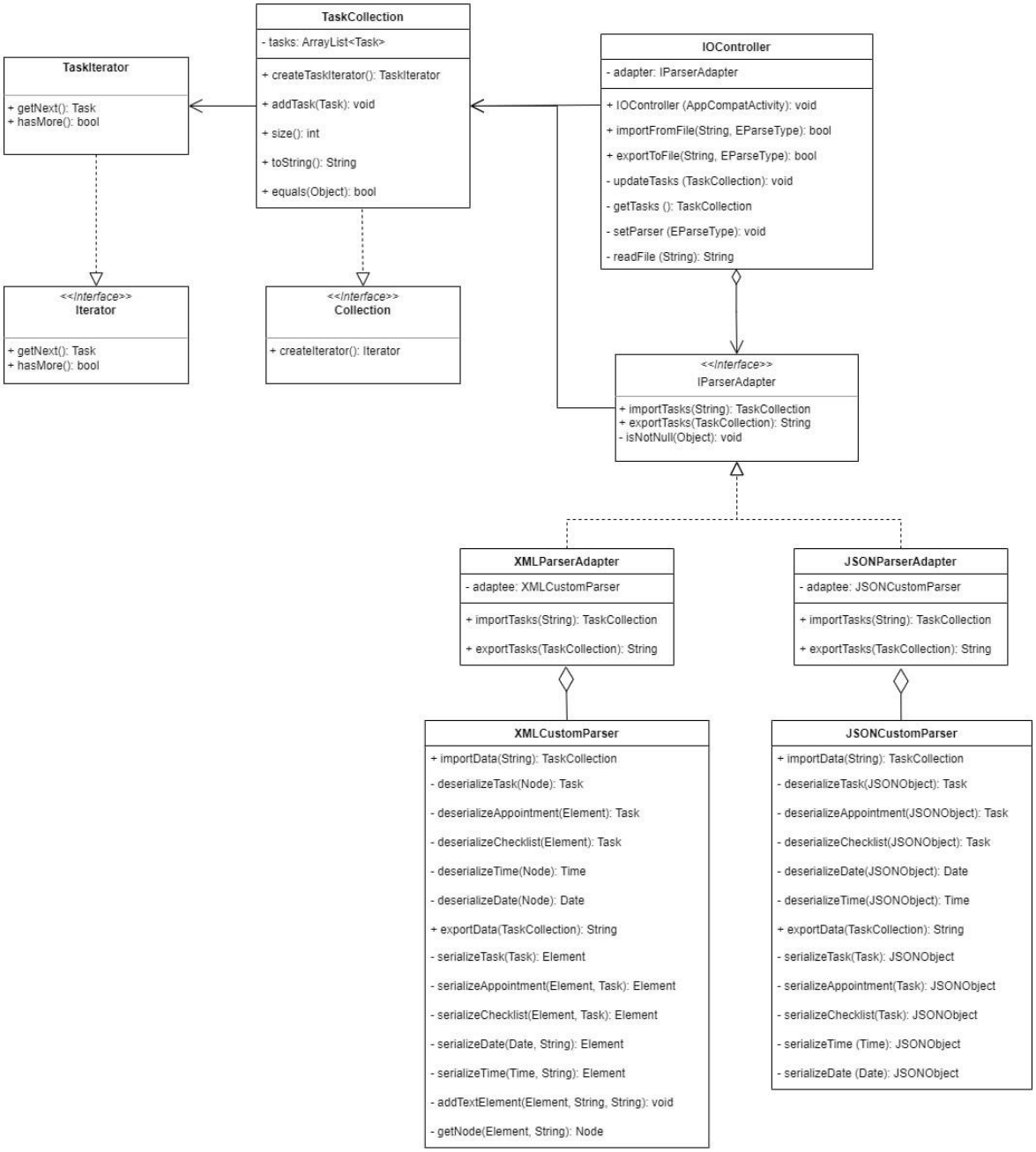
Model: TaskListCalendarView responsible for saving in DB chosen colour for the task, the updated task position and chosen view.



Member 5 responsibility

Task management application has to support import/export of tasks. The export function supports two serialization file formats JSON and XML. The import function reads task list from a serialization file and adds them to the database.

Design Patterns: Iterator, Adapter.



All of the classes on the UML diagram above are part of the Model in MVVM archetecture. The diagram will probably be extended in the future with classes in the View part, as we will need a separete visualization for export/import functions.

IOController provides functionality to import and export a Task. Task will be imported to the database from file or exported from the database to file.

Iterator pattern:

TaskCollection provides class for storing a Task in dedicated object. It implements interface *IContainer*, that was renamed from *Collection* during implementation phase, as such class already exists in java.

Class *TaskIterator* implements interface *Iterator* and is used to actually iterate the collection of tasks we that are to be imported/exported.

Adapter pattern:

Interface *IParserAdapter* is a single parser adapter interface. Interface implemented by two different parser adapters *XMLParserAdapter* and *JSONParserAdapter*.

XMLParserAdapter class is dedicated to adapt *XMLParser* to *IParserAdapter* interface to specify on XML files.

JSONParserAdapter class is dedicated to adapt *JSONParser* to *IParserAdapter* interface to specify on JSON files.

XMLCustomParser and *JSONCustomParser* classes are dedicated to parse XML and JSON formatted strings. Contain methods to serialize and deserialize these string.

1.1.2 Technology Stack

- Android App project setup and development:
  - IDE for android app development: Android Studio
  - API level: Android 9.0 (Pie, API Level 28)
  - Libraries: Android Jetpack
  - Android virtual device: Phone Pixel 2, Release: Q (API Level 29), Target: Android 10.0
- Programming language: Java
  - Java version: 1.8 (Java 8)
- Build tool: Gradle
  - Gradle version: 7.4
- SDK Versions
  - Compile SDK Version: 32
  - Min SDK Version: 28
  - Target SDK Version: 32
- Database: SQLite
- ORM library: Room
  - Room version: 2.4.3
- Testing library: JUnit
  - JUnit version: 4.13.2
- Email Testing tool: https://mailtrap.io

In our implementation we use Android Jetpack libraries, which contain useful in-built features for developing and make our application executable consistently across

different Android versions. Java is our main programming language, as it is very efficient for developing Android applications and our team already had experience with it in the past. For managing project dependencies, building and running our application we make use of the Gradle build automation tool.

We decided to use SQLite as a database because it is lightweight and can be used on an Android application built on Java. To access and persist data in the database we are using the Room library, which provides an abstraction layer over SQLite.

For testing purposes we are going to use the JUnit library, which provides a lot of useful and efficient functionalities for testing application.

According to the project developing recommendations, we use API level 28 of Android 9.0 Pie and Phone Pixel 2 as Android virtual device.

1.2 Major Changes Compared to DESIGN

Member 1 Part:

Another subclass, namely Checklist, was introduced. For that the Dao, Proxy and TaskViewModel had to be adapted properly.

The Recycler View now had to be able to hold 2 different kinds of item views in one View. Also there were different Views needed for the creation and updating for each Task class.

An Exception class was also implemented. The NoTitleException which handles an empty title in the creation of a Task. A toast will let the user know to enter a title, otherwise it won't be created.

Furthermore several methods were introduced, which are then being used in various situations to ensure single responsibility.

Member 3 Part:

NotificationViewModel class was added which is responsible for showing pop-up notifications and initialising ScheduledUpcomingAppointmentChecker with application context.

New PopUpNotificationMesageHandler was introduced to fulfil requirements for sending pop-up notifications. This class has a LiveData field which stores the information which should be shown to the user in pop-up. PopUpNotification class writes data to this field and notificationViewModel observes it and shows pop-up if needed.

Utility class EmailValidator was added to do email validations on UI.

In addition more custom exception classes were added to improve exception handling.

Member 5 Part:

No major architectural changes were made from the DESIGN phase. Main changes are connected with creation of helping methods to increase cohesion. We also created an exception class ParsingExeption to unify exception handling.

1. IOController class

   Create updateTasks (TaskCollection): void. Layer of abstraction to collect the tasks for import.

   Create getTasks (): TaskCollection. Layer of abstraction to collect the tasks for export.

   Rename determineFormat(Type) to setParser (EParseType): void. To be able to define file type by buttons, so that user does not need to type it.

   Create readFile (String): String. To separate reading file functionality from import method.

   Pass AppCompatActivity to IOController class to be able to work with database.

2. JSONCustomParser:

   Create deserializeTask(JSONObject): Task. To separate deserealizing functionality for task from import method

   Create deserializeAppointment(JSONObject): Task. To separate deserealizing functionality for appointment from import and deserializeTask methods.

   Create deserializeChecklist(JSONObject): Task. To separate deserealizing functionality for checklist from import and deserializeTask methods.

   Create deserializeDate(JSONObject): Date. To separate deserealizing functionality for data from import method.

   Create deserializeTime(JSONObject): Time. To separate deserealizing functionality for time from import method.

   Create serializeTask(Task): JSONObject. To separate serealizing functionality for task from export method.

Create serializeAppointment(Task): JSONObject. To separate serealizing functionality for appointment from export and serializeTask methods.

Create serializeChecklist(Task): JSONObject. To separate serealizing functionality for checklist from export and serializeTask methods.

Create serializeTime (Time): JSONObject. To separate serealizing functionality for time from export and serializeTask methods.

Create serializeDate (Date): JSONObject. To separate serealizing functionality for date from export and serializeTask methods.

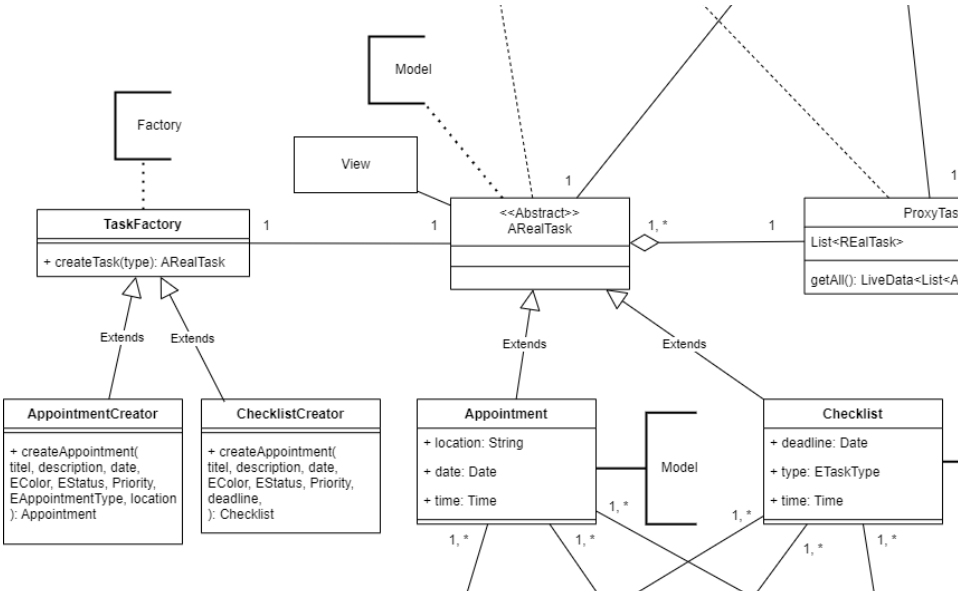
3. XMLCustomParser had similar changes as JSONCustomParser.

1.3 Design Patterns

1.3.1 Factory Pattern

The Factory Pattern will be used in the creation of the Subclasses for the Task class. This pattern is based upon the inheritance property of an OOP Language. By having a Factory we want to have a dynamical and flexible creation of objects.

A common superclass is needed for this. The subclass is going to be instantiated to an Object of the Superclass. Like this it is easy to introduce new subclasses to the code without changing a lot of code.




```
Task toBeInserted = new Checklist(title, descript
taskViewModel.insert(toBeInserted);
```

```
Task toBeInserted = new Appointment(title, desc
taskViewModel.insert(toBeInserted);
```

Factory Pattern for Notification functionality:

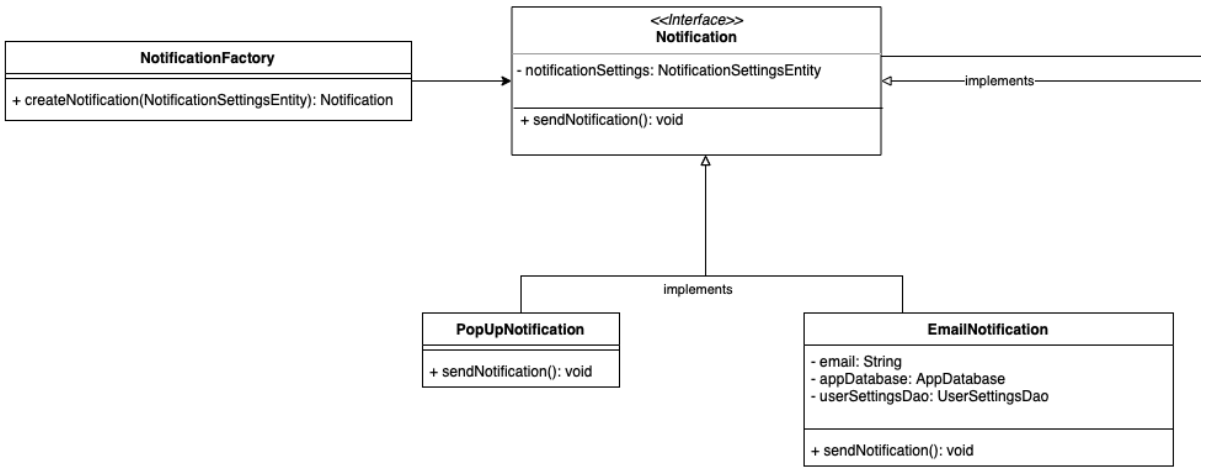
The Factory Pattern is the approach which helps us to flexibly create new objects without directly instantiating these objects.

In our implementation this pattern is used in order to create the Notification class. As we have two notification types Email and Popup, sending notifications in these classes will be implemented in different ways. And the main idea is to hide this logic and simplify the class creation process.

Also advantage of using Factory, which can solve some of the problems in the future by extending and adding new notification type (e.g SMS, Push etc.) to our implementation, is that we just have to add the new subclass with the *sendNotification()* method for the new type of notification.

Factory pattern consists of:

- *Notification* interface
- *PopUpNotification* and *EmailNotification* classes that implement *Notification* interface with the logic for sending notifications and contain necessary fields.
- *NotificationFactory* class is responsible for creating the notification classes.



```java
public class NotificationFactory {
    public Notification createNotification(NotificationSettings notificationSettings){
        if (notificationSettings == null)
            return null;
        switch (notificationSettings.notificationType) {
            case POPUP:
                return new PopUpNotification();
            case EMAIL:
                return new EmailNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type");
        }
    }
}
```

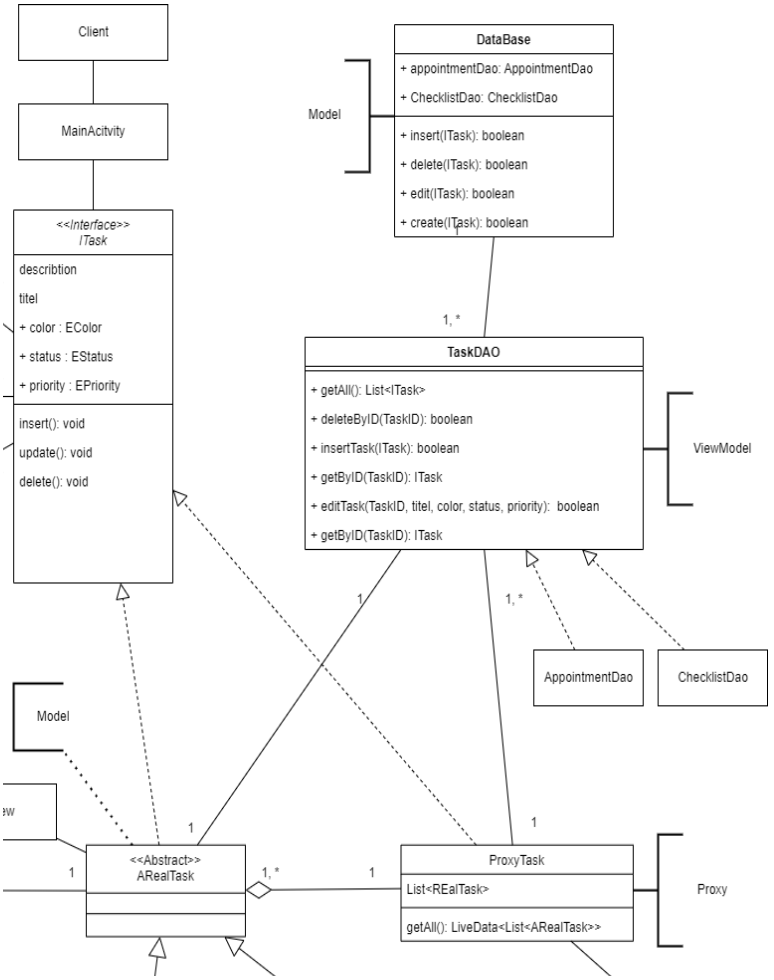
```java
public class PopUpNotification implements Notification{
    @Override
    public void sendNotification(){}
}
```

```java
public class EmailNotification implements Notification{

    @Override
    public void sendNotification(){}
}
```

1.3.2 Proxy Pattern

With the Proxy Pattern one wants to create a mediator who interacts with the Client/Program and controls data access. This can help to ensure how data is being accessed and creates an extra security layer.

We are planning to implement this pattern for our Task. We will have an instance ProxyTask which will hold all of the created Tasks, control their creation, access, storing and various manipulations of the data.



As illustrated in this UML snippet there is an instance of a real Task class and an instance of the Proxy Task class. Both inherit from the Interface ITask. Hence all the methods used in the real Task have to be implemented in the Proxy as well. Like this we make sure that all we want to do in the real Task will also be possible through the Proxy. Furthermore the Proxy will provide additional functionality (e.g. fetching all Tasks which have been created and saved in the database)

```
public class TaskProxy implements ITask {
    private AppDatabase database;
    private static TaskProxy instance;
    private TaskDao taskDao;

    public static TaskProxy getInstance(final AppDatabase database){
        if (instance == null) {
            synchronized (TaskProxy.class) {
                if (instance == null) {
                    instance = new TaskProxy(database);
                }
            }
        }
        return instance;
    }

    private TaskProxy(AppDatabase db){
        this.database = db;
        taskDao = database.taskDao();
        observers.add(new NotificationObserver(db));
    }
```

```java
// https://developer.android.com/reference/android/arch/lifecycle/MediatorLiveData
public LiveData<List<Task>> getAll() {
    MediatorLiveData<List<Task>> result = new MediatorLiveData<>();
    result.setValue(new ArrayList<>());
    List<Task> allTasksList = new ArrayList<>();

    LiveData<List<Appointment>> appointments = taskDao.getAllAppointments();
    LiveData<List<Checklist>> checklists = taskDao.getAllChecklists();

    result.addSource(appointments, tasks -> {
        allTasksList.clear();
        allTasksList.addAll(tasks);
        if(checklists.getValue() != null)
            allTasksList.addAll(checklists.getValue());
        result.setValue(allTasksList);

    });

    result.addSource(checklists, tasks -> {
        allTasksList.clear();
        if(appointments.getValue() != null)
            allTasksList.addAll(appointments.getValue());
        allTasksList.addAll(tasks);
        result.setValue(allTasksList);
    });

    return result;
}
```

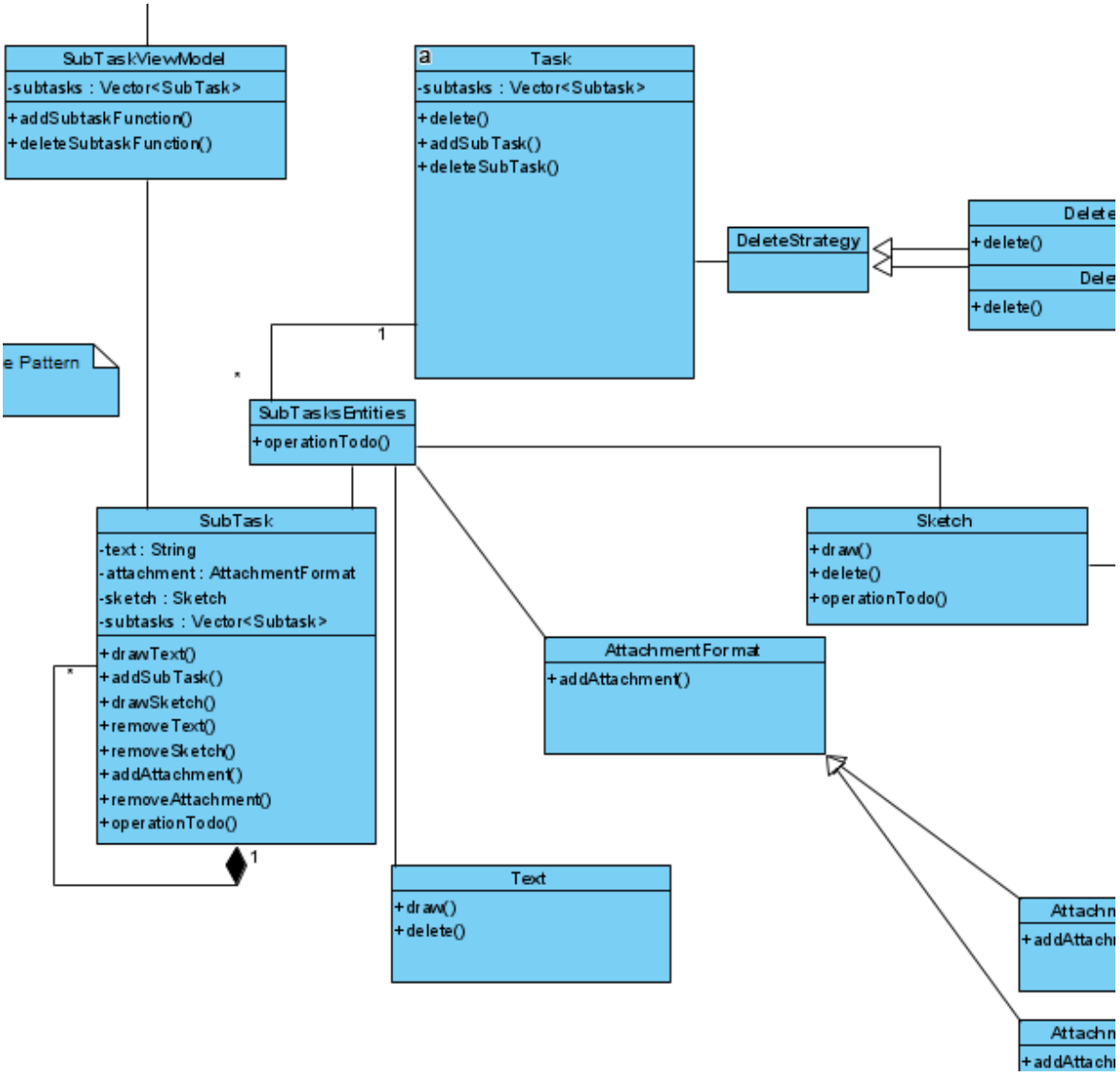
```
public LiveData<List<Appointment>> getAppointments(){ return taskDao.getAllAppointments();}

@Override
public void insert(Task task) {
    if (task instanceof Appointment) {
        Appointment appointment = (Appointment) task;
        AppDatabase.databaseWriteExecutor.execute(() -> {
            taskDao.insertAppointment(appointment);
            Log.d(TAG,  msg: "TaskProxy insert() returned: " + "appointment saved");
        });
    }
    if(task instanceof Checklist) {
        Checklist checklist = (Checklist) task;
        AppDatabase.databaseWriteExecutor.execute(() -> {
            taskDao.insertChecklist(checklist);
            Log.d(TAG,  msg: "TaskProxy insert() returned: " + "checklist saved");
        });
    }

    notifyAllObservers(ActionType.TASK_CREATED, task);
}
```

```java
@Override
public void update(Task task) {
    if(task.getSubclass().equals("Appointment")){
        AppDatabase.databaseWriteExecutor.execute(() -> {
            Log.d(TAG, msg: "update() TaskProxy returned: " + "class is instance of Appointment.");
            taskDao.updateAppointment((Appointment) task);
        });
    }else if(task.getSubclass().equals("Checklist")){
        AppDatabase.databaseWriteExecutor.execute(() -> {
            Log.d(TAG, msg: "update() TaskProxy returned: " + "class is instance of Checklist.");
            taskDao.updateChecklist((Checklist) task);
        });
    }else{
        Log.d(TAG, msg: "update() TaskProxy returned: " + "class no instance of anything known.");
    }
    notifyAllObservers(ActionType.TASK_UPDATED, task);
}

@Override
public void delete(Task task) {
    if(task.getSubclass().equals("Appointment")){
        AppDatabase.databaseWriteExecutor.execute(() -> {
            Log.d(TAG, msg: "delete() TaskProxy returned: " + "class is instance of Appointment.");
            taskDao.deleteAppointment((Appointment) task);
        });
    }else if(task.getSubclass().equals("Checklist")){
        AppDatabase.databaseWriteExecutor.execute(() -> {
            Log.d(TAG, msg: "delete() TaskProxy returned: " + "class is instance of Checklist.");
            taskDao.deleteChecklist((Checklist) task);
        });
    }else{
        Log.d(TAG, msg: "delete() TaskProxy returned: " + "class no instance of anything known.");
    }
    notifyAllObservers(ActionType.TASK_DELETED, task);
}
```

```java
public void deleteAll() {
    AppDatabase.databaseWriteExecutor.execute(() -> {
        Log.d(TAG, msg: "delete() TaskProxy returned: " + "class is instance of Checklist.");
        taskDao.deleteAllFromChecklist();
        taskDao.deleteAllFromAppointment();
    });
}
```

1.3.3 Composite Pattern

- At least one type of task should be able to be composed of subtasks
- We need to create a system, which contains Tasks, that contains Subtasks, Attachment etc. which can contain Subtask and all functions of previous Subtask.



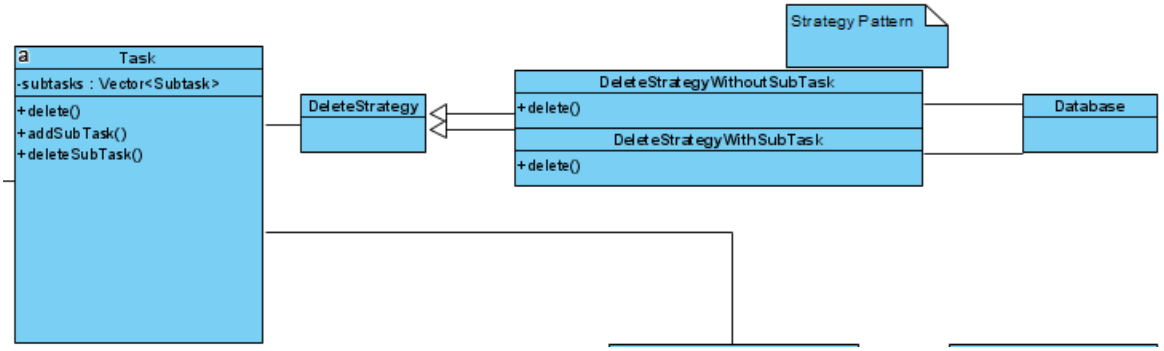

```java
package com.example.se2_team06.model;

public interface SubTasksEntities {
    public void operationTodo();
}
```

Interface

```java
package com.example.se2_team06.model;

public class SubTask implements SubTasksEntities{
    private final AttachmentFormat attachment;
    private final Sketch sketch;

    public SubTask(AttachmentFormat attachment, Sketch sketch){
        this.attachment = attachment;
        this.sketch = sketch;
    }
    public void addAttachment(){

    }
    public void deleteAttachment(){

    }
    public void addSketch(){

    }
    public void deleteSketch(){

    }
    public void addText(){

    }

    @Override
    public void operationTodo() {

    }
```

```java
package com.example.se2_team06.model;

public class Sketch implements SubTasksEntities{
    public Sketch(){

    }
    public void draw(){

    }
    5 related problems
    public void delete(){

    }

    @Override
    public void operationTodo() {

    }
}
```

1.3.4 Strategy Pattern

- A suitable strategy for deletion of parent and/or subtasks should be implemented (e.g. how to handle the deletion of a task that contains subtasks).
- We should develop an algorithm to delete tasks and subtasks.
- 

  Tasks have 2 Delete Strategy: one is for deleting only the main task and second one is for cases when the Task has at least one subtask in it. If Second one is called, it will delete the whole cascade of Subtasks.

-
```java
public interface DeleteStrategy {
    public void delete();
}
```

```java
package com.example.se2_team06.model;

public class DeleteStrategyWithoutSubTask implements DeleteStrategy{
    @Override
    public void delete() {
        //Delete without subtasks
    }
}
```

```java
package com.example.se2_team06.model;

public class DeleteStrategyWithSubTask implements DeleteStrategy{
    @Override
    public void delete() {
        //Delete with subtasks
    }
}
```

1.3.5 Observer Pattern

In our implementation we decided to use the Observer pattern for notification sending related functionality. This pattern provides a solution for the case, when application user should be notified, when a task is created, updated, deleted and when an appointment is coming up.

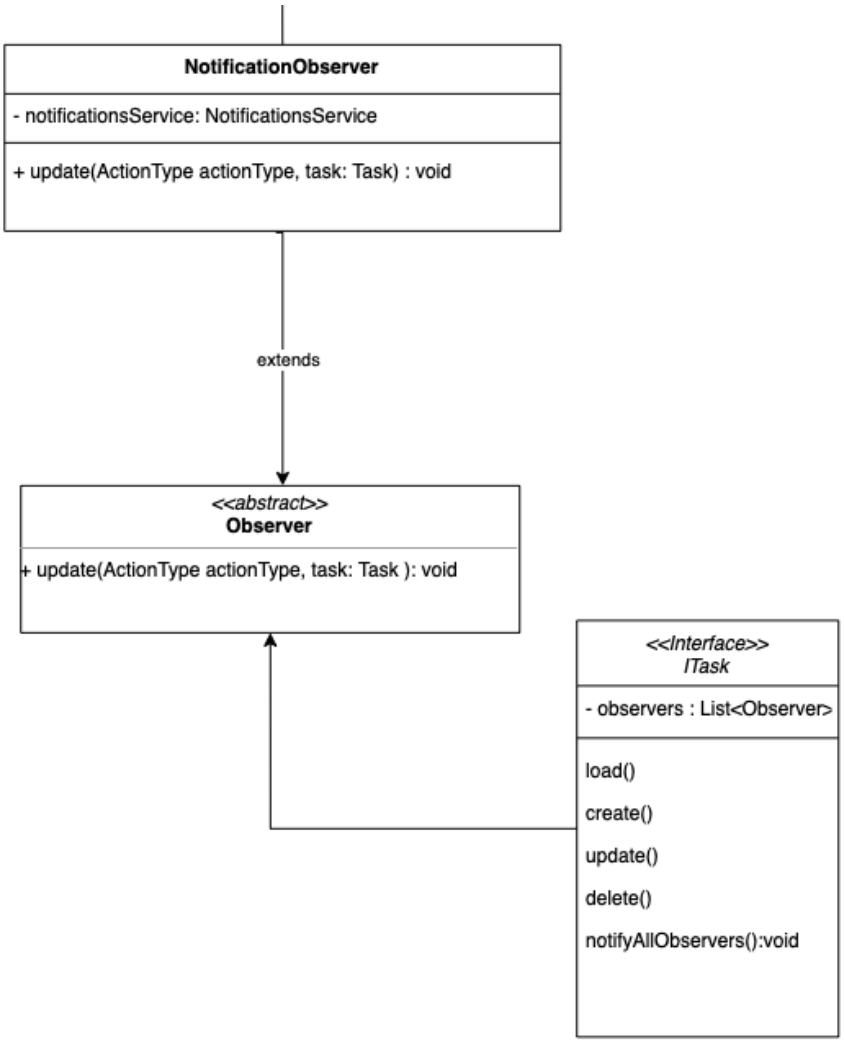
The problem, that the Observer pattern solves, is that the notification service doesn't need to check the state of the task to send a notification, instead everything proceeds automatically. And also to prevent the user from not being notified of any action.

Observer Pattern components are: abstract class *Observer,* which is parent and contains *update*() method; *NotificationObserver,* which extends the abstract class;

ITask class, which should be observed, has a list of observers and method *notifyAllObservers()*; *NotificationService,* which will be used from *NotificationObserver* to send a notifications.

List of observers in *ITask* class will be notified when a task is created, updated or deleted. This list of observers can easily be extended with new observer classes if needed in the future.

NotificationObserver will be automatically notified, when the state of the tasks is changed, and in its turn will notify the *NotificationService* class to send a notification.



```java
public interface ITask {
    List<Observer> observers = null;

    void insert(Task task);
    void update(Task task);
    void delete(Task task);
    void notifyAllObservers();
}
```

```java
public abstract class Observer {
    public void update(ActionType actionType, Task task){}
}
```

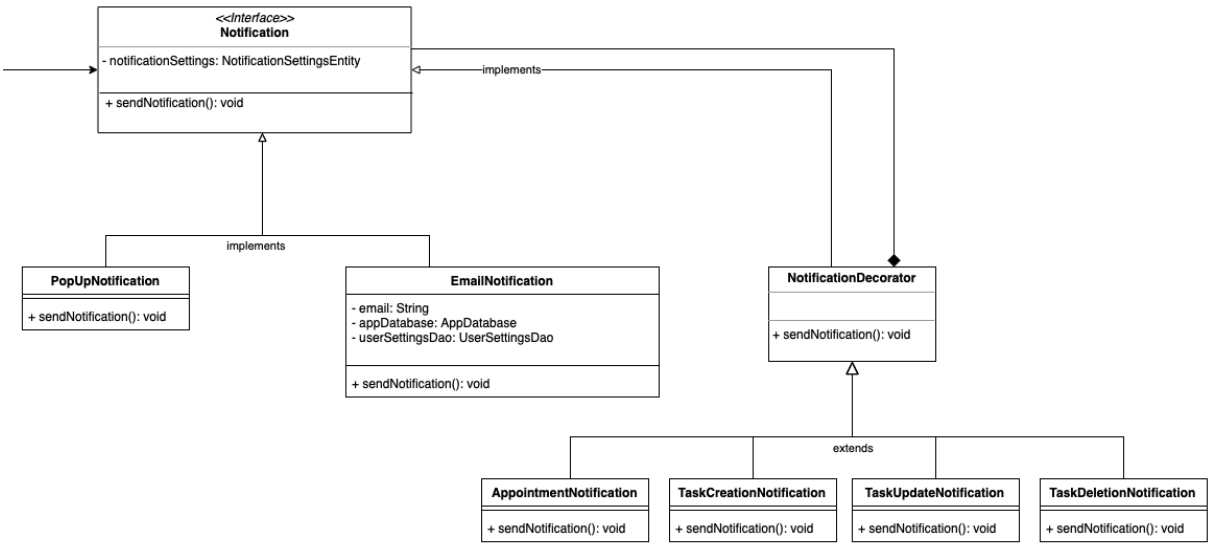
```java
public class NotificationObserver extends Observer{
    private NotificationService notificationService;

    @Override
    public void update(ActionType actionType, Task task) { super.update(actionType, task); }
}
```

1.3.6 Decorator Pattern

Decorator pattern is used to decorate Email and Pop-up Notifications according to the action. We have four different actions, about which the user needs to be notified: when a task is created, updated, deleted or an appointment is coming up. For each of these actions notification should have different and customizable content. Using decorator classes, the actions to be notified can be modified and used independently. Also the list of decorators can be easily extended by adding new decorator classes without affecting the rest of the logic. The Decorator pattern also provides the ability to receive notifications through various channels, in our case, email as well as pop-ups.

Decorator Pattern consists of:

- *Notification* interface with *sendNotification()* method, which is implemented by the *PopUpNotification* and *EmailNotification* classes.
- an abstract class *NotificationDecorator,* which also enhances *Notification* interface and is parent for all of Decorator classes.
- Decorator classes: *AppointmentNotification*, *TaskCreationNotification*, *TaskUpdateNotification*, *TaskDeletionNotification,* which extend the *NotificationDecorator* and modify the *sendNotification()* method according to the action type.




```java
public interface Notification {
    NotificationSettings notificationSettings = null;

    void sendNotification();
}
```

```java
public class EmailNotification implements Notification{

    @Override
    public void sendNotification(){}
}
```

```java
public class PopUpNotification implements Notification{
    @Override
    public void sendNotification(){}
}
```

```java
public class NotificationDecorator implements Notification{
    protected Notification notification;

    public NotificationDecorator(Notification notification) { this.notification = notification; }

    @Override
    public void sendNotification() { this.notification.sendNotification(); }
}
```

```java
public class AppointmentNotification extends NotificationDecorator{

    public AppointmentNotification(Notification notification) { super(notification); }

    @Override
    public void sendNotification() { super.sendNotification(); }
}
```

```java
public class TaskCreationNotification extends NotificationDecorator{

    public TaskCreationNotification(Notification notification) { super(notification); }

    @Override
    public void sendNotification() { super.sendNotification(); }
}
```

```java
public class TaskDeletionNotification extends NotificationDecorator{

    public TaskDeletionNotification(Notification notification) { super(notification); }

    @Override
    public void sendNotification() { super.sendNotification(); }
}
```

```java
public class TaskUpdateNotification extends NotificationDecorator{

    public TaskUpdateNotification(Notification notification) { super(notification); }

    @Override
    public void sendNotification() { super.sendNotification(); }
}
```

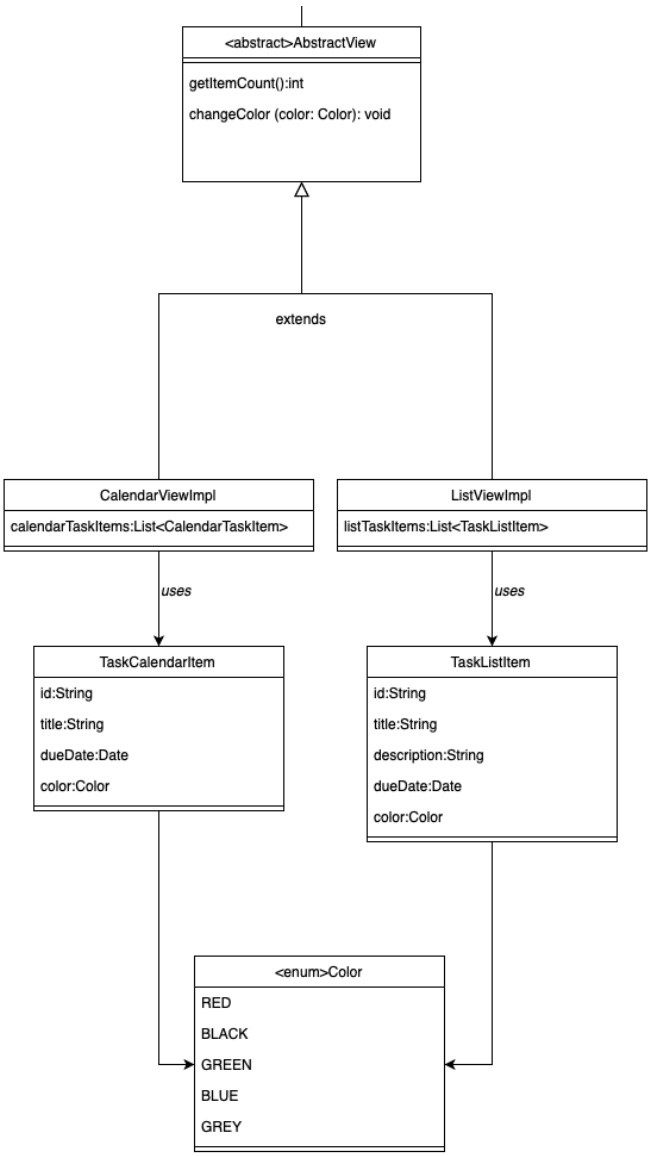
1.3.7 Facade Pattern

Facade pattern approach:

Facade is a design pattern which provides a simplified interface to a complex system of sub-classes.
The "AbstractView" class has following functions:

- returns number of items;
- changes the colour of an element of an calendar and list views.

In our case the Facade provides access to CalendarViewImpl and ListViewImpl classes.




```java
import android.graphics.Color;

public abstract class AbstractView {
    public abstract int getItemCount();
    public abstract int changeColor(EColor color);
}
```

```java
public class CalendarViewImpl extends AbstractView{
    private List<CalendarTaskItem> calendarTaskItems;

    @Override
    public int getItemCount() { return 0; }

    @Override
    public int changeColor(EColor color) { return 0; }
}
```

```java
import java.util.List;

public class ListViewImpl extends AbstractView{
    private List<TaskListItem> taskListItems;

    @Override
    public int getItemCount() { return 0; }

    @Override
    public int changeColor(EColor color) { return 0; }
}
```

```java
public class TaskListItem {
    private String id;
    private String title;
    private String description;
    private Date dueDate;
    private EColor eColor;
}
```

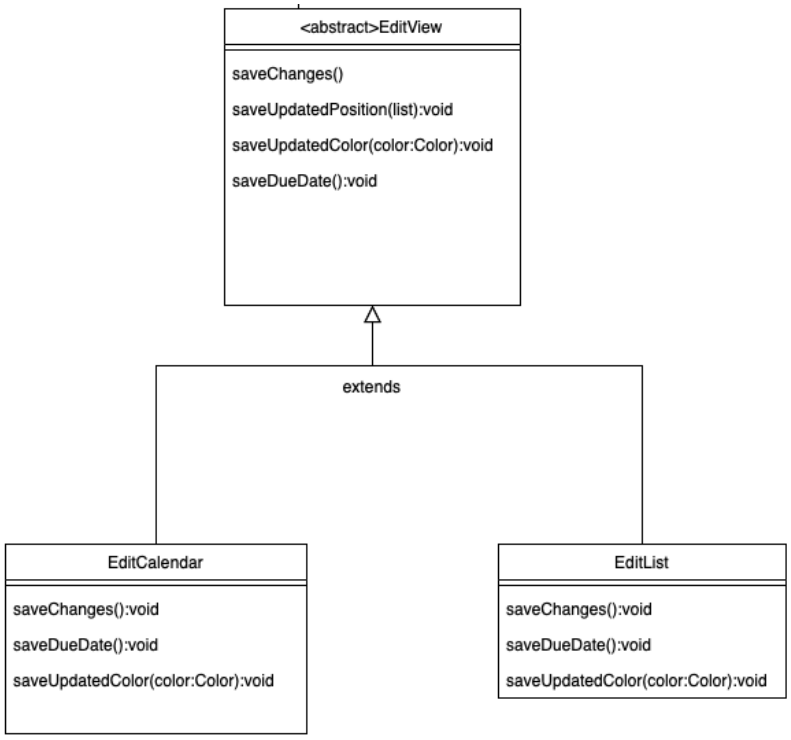
```java
public class CalendarTaskItem {
    private String id;
    private String title;
    private Date dueDate;
    private EColor eColor;
}
```

```java
public enum EColor {
    RED( friendlyName: "Red"),
    BLACK( friendlyName: "Black"),
    GREEN( friendlyName: "Green"),
    BLUE( friendlyName: "Blue"),
    GREY( friendlyName: "Grey");

    private String friendlyName;

    EColor(String friendlyName) { this.friendlyName = friendlyName; }

    @Override public String toString() { return friendlyName; }
}
```

1.3.8 Template method

Template method approach:

Template method is a design pattern which defines the skeleton of an algorithm in the superclass, which lets the client to override only certain classes of a complex system. In our particular case the abstract class EditView defines the steps for editing calendar and list views, the steps are overridden in subclasses for each view.




```
public abstract class EditView {
    public void saveChanges() {

    }

    public void saveUpdatedPosition() {

    }

    public void saveUpdatedColor(EColor color) {

    }

    public void saveDueDate() {

    }
}
```

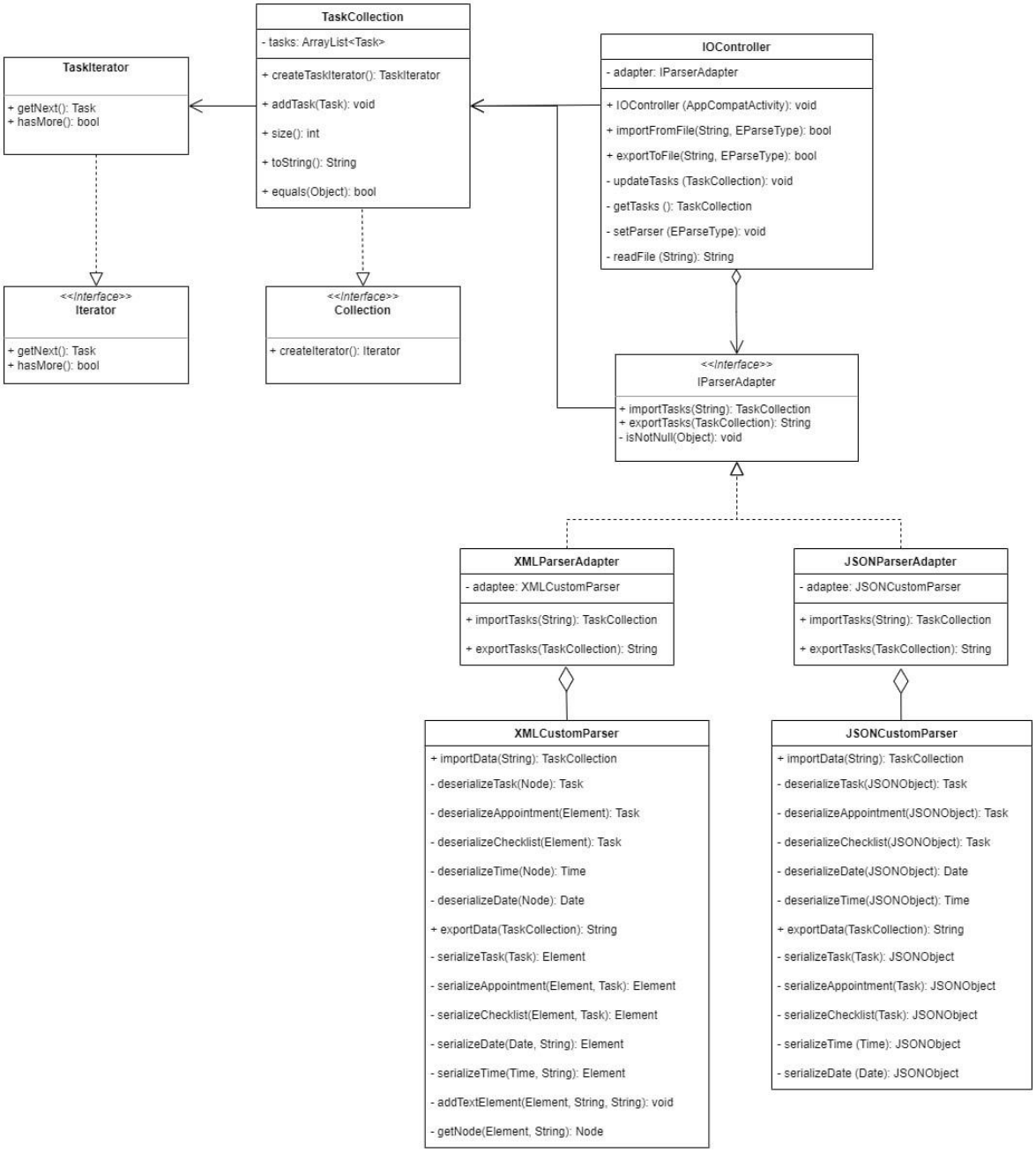
```java
public class EditCalendar extends EditView {
    @Override
    public void saveChanges() { super.saveChanges(); }

    @Override
    public void saveUpdatedPosition() { super.saveUpdatedPosition(); }

    @Override
    public void saveUpdatedColor(EColor color) { super.saveUpdatedColor(color); }

    @Override
    public void saveDueDate() { super.saveDueDate(); }
}
```

```java
public class EditList extends EditView{
    @Override
    public void saveChanges() { super.saveChanges(); }

    @Override
    public void saveUpdatedPosition() { super.saveUpdatedPosition(); }

    @Override
    public void saveUpdatedColor(EColor color) { super.saveUpdatedColor(color); }

    @Override
    public void saveDueDate() { super.saveDueDate(); }
}
```

1.3.9 Adapter Pattern

Adapter pattern lets objects with different interfaces to collaborate.

In our application is used to work with and parse JSON and XML files.



XMLCustomParser and *JSONCustomParser* classes are the two adaptees of this pattern. They are dedicated to parse XML and JSON formatted strings. Contain methods to serialize and deserialize these string. Below is skeleton of *JSONCustomParser.*

```
/**
 * Class dedicated to parsing JSON-formatted string.
 * Contains methods to serialize and deserialize JSON-formatted string.
 */
public class JSONCustomParser {
```

```
    /**
     * Methods deserializes collection of tasks
     * into Java native class from JSON-formatted string.
     * @param string JSON-formatted string.
     * @return Collection of tasks deserialized from passed string.
     */
    public TaskCollection importData(String string) {
        TaskCollection tasks = new TaskCollection();
        Object obj = null;
        try {
            obj = new JSONParser().parse(string);
        } catch (ParseException e) {
            System.out.println("ParseException");
            return tasks;
        }
        JSONObject jo = (JSONObject) obj;
        // Resolve list of task.
        JSONArray ja_tasks = (JSONArray) jo.get("tasks");
        if (ja_tasks == null) {
            System.out.println("ParseException");
            return tasks;
        }
        for (Object task : ja_tasks) {
            if (!(task instanceof JSONObject)) {
                System.out.println("ParseException");
                return tasks;
            }
            JSONObject jo_task = (JSONObject) task;
            tasks.addTask(this.deserializeTask(jo_task));
        }
        return tasks;
    }
```

```
    /**
     * Method parses task from 'jo' object
     * into Java native class.
     * @param jo Object, that contains serializes task.
     * @return Task, parsed from 'jo' object.
     */
    private Task deserializeTask(JSONObject jo) {
        return new Task(
                "test_task",
                "simple_test_task",
                EColor.BLACK,
                EStatus.PLANNED,
                EPriority.LOW
        );
    }
    /**
     * Method serializes collection of tasks
     * from Java native class into JSON-formatted string.
     * @param tasks Collection of tasks to be serialized.
     * @return JSON-formatted string, that contains serialized tasks.
     */
    public String exportData(TaskCollection tasks) {
        return "";
    }
    /**
     * Method serializes task from Java native class
     * into dedicated JSON object.
     * @param task Task to be serialized.
     * @return JSON object, that contains serialized task.
     */
    private JSONObject serializeTask(Task task) {
    return new JSONObject();
```

```
}}
```

To help parsing without caring about the format of the file, we use the Interface *IParserAdapter,* which is implemented by two different parser adapters, *XMLParserAdapter* and *JSONParserAdapter,* accordingly.

```
package com.example.se2_team06;

/**
 * Single parser adapter interface.
 * Interface implemented by different parser adapter.
 */
public interface IParserAdapter {

    public TaskCollection importTasks (String string);
    public String exportTasks (TaskCollection tasks);

}
```

XMLParserAdapter class is dedicated to adapt *XMLParser* to *IParserAdapter* interface to specify on XML files.

JSONParserAdapter class is dedicated to adapt *JSONParser* to *IParserAdapter* interface to specify on JSON files.

```
package com.example.se2_team06;

/**
 * Class dedicated to adapt JSONParser to IParserAdapter interface.
 */

public class JSONParserAdapter implements IParserAdapter {

    final private JSONCustomParser adaptee = new JSONCustomParser();

    public TaskCollection importTasks (String string) {
        return this.adaptee.importData(string);
    }

    public String exportTasks (TaskCollection tasks) {
        return this.adaptee.exportData(tasks);
    }

}
```

```java
package com.example.se2_team06;

/**
 * Class dedicated to adapt XMLParser to IParserAdapter interface.
 */

public class XMLParserAdapter implements IParserAdapter {

    final private XMLCustomParser adaptee = new XMLCustomParser();

    public TaskCollection importTasks(String string) {

        return this.adaptee.importData(string);
    }

    public String exportTasks (TaskCollection tasks) {
        return this.adaptee.exportData(tasks);
    }

}
```

IOController is used to implement the basic functionality of export/import of tasks.

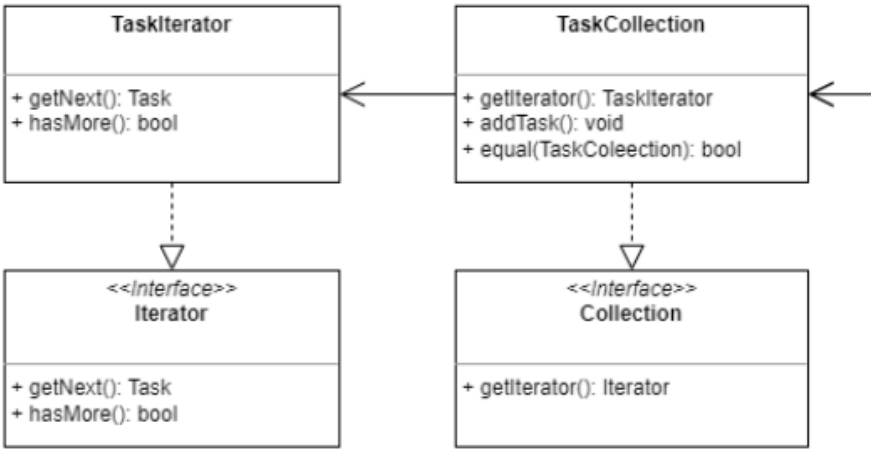
```
public class IOController {

   private IParserAdapter adapter;

   /**

    * Import tasks from file with filename 'string' to the database.

    * @param string Name of file, that contains serialized tasks.

    * @return True if operation is successful else false.

    */

   public boolean importFromFile(String string) {

      return true;

   }

   /**

    * Export tasks to file with filename 'string' from the database.

    * @param string Name of file, that will contains serialized tasks.

    * @return True if operation is successful else false.

    */

   public boolean exportToFile(String string) {

      return true;

   }
```

```
    /**
     * Method determines the format of string to be deserialized from or
     * to be serialized to. Creates either JSON or XML parser and
     * assign it to class attribute 'adapter'. As a result,
     * other methods are not aware of type of parser,
     * that they are using.
     * @param string Formatted string, whose format is determined.
     */
    private void determineFormat(String string) {  }
}
```

1.3.10 Iterator Pattern

Iterator pattern is used to traverse collection elements without paying attention at its underlying representation e.g. list, tree, etc.

In our application we use it to iterate task list that is to be exported or imported from the database. Specifically, to read tasks from *TaskCollection*.



TaskCollection provides class for storing a Task in dedicated object. It implements interface *IContainer*, that was renamed from *Collection* during implementation phase, as such class already exists in java.

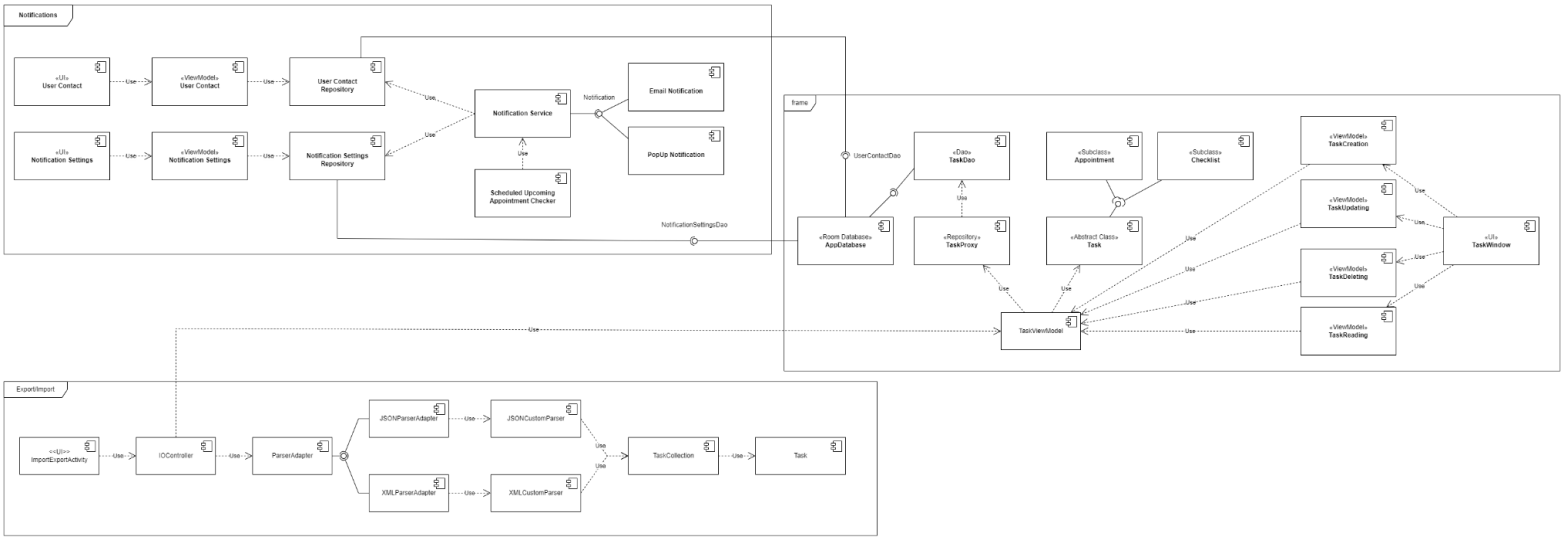
```java
package com.example.se2_team06;

public interface IContainer {

    public Iterator<Task> getIterator();

}
```

Class *TaskIterator* implements interface *Iterator* and is used to actually iterate the collection of tasks we that are to be imported/exported.

```java
package com.example.se2_team06;

public interface Iterator <T> {

    boolean hasNext();
    T next();

}

public class TaskCollection implements IContainer{

        private ArrayList<Task> tasks;

        public Iterator<Task> getIterator() {
                return new TaskIterator();
        }

        public void addTask(Task task) {

        }

        private class TaskIterator implements Iterator<Task> {

                private int index = 0;

                @Override
                public boolean hasNext() {
                        return index < tasks.size();
                }

                @Override
                public Task next() {
                        if (this.hasNext()) {
                                return tasks.get(this.index++);
                        }
                        return null;
                }
        }
}
```

2 Implementation

2.1 Overview of Main Modules and Components



2.2 Quality Requirements Coverage

This will be discussed for each member in the following chapters(1.1.1, 2.3, 2.4, 2.5, 2.6)

2.3 Coding Practices

Member 1:

The naming of the methods/functions were always chosen to be clear and understandable. For this reason there was not much use of comments, because the naming of everything is mostly self-explanatory. Classes have a noun as name, functions a verb.

The few comments that were made follow the commenting style presented in the slides of the lecture

(/**

* comment

*/)

For better maintainability and readability the functions were held short, by introducing single responsibility with the above mentioned naming conventions.

Also the formatting was done by the IDE.

Most importantly I've been consistent throughout the programming phase, which eliminates potential errors in understanding.

Member 3:

To ensure code readability and maintainability during implementation the following steps were taken into account:

- Java coding conventions were followed
- Naming of variables were chosen to represent the data which is stored in this variable: for example a variable which contains a string with an email address for an email receiver named: *emailReceiverAddress*.
- Naming of functions were chosen to represent the process which this function does for example: *sendNotification()*.
- Preventing redundant comments with proper naming of functions, classes and variables.
- For formatting, the IDE settings were set to automatically format the code.
- Each method is responsible for one functionality to reduce code complexity.

Member 5:

To support readability and maintainability of code the following coding techniques were applied:

- Following the naming rules from the lectures, e.g. verb containfing functions names and noun-phrase object/classes names.
- Names are meaningful and searchable.
- Comments where used for complex functions, in case using proper naming was not enough.
- Formatting was performed automatically in IDE.
- Tried to increase cohesion of the methods.

2.4 Key Design Principles

Member 1:
Abstraction was used to implement the different Task types which inherit from one Superclass.

Single Responsibility was used widely, having every checking or functionality in a separate function. and only one functionality. (e.g. checktitle(title) in the saveTask() function, its not the responsibility of saveTask to check the input from the user is valid, thats being outsourced)

High Cohesion One class does exactly one thing, and everything serves the same purpose (AddChecklistActivity, AddTaskActivity, …)

Low Coupling was implemented that functions/methods were not dependent on other components of the program. The creation of a Task passes the parameters back to the main activity, which holds the TaskViewModel to then insert it to the database. Creation is not coupled tightly with the TaskViewModel class.

Level of Indirectness, with the Proxy Pattern I was able to create a level of indirectness. This means before executing operations on an item I was able to check the given parameters and either allow or deny the operation, or even add some more functionality, parameters ect.

Member 3:

Hierarchy was used to represent different notification classes, where we have parent abstract class Notification.java and its children PopUpNotification.java and EmailNotification.java. By using hierarchy new notification type implementations can be easily added.

Abstraction was implemented to use polymorphism in our code e.g. by sending notifications without defining the specific notification type. Notification.java is an abstract class which is extended by PopUpNotification.java and EmailNotification.java classes.

In the example below, the notification variable of abstract class Notification was created then initialised this variable with concrete notification in createNotification() method. After that we call sendNotification() of our notification object, although we do not know which type of Notification class exactly it is.

```
Notification notification;
notification = notificationFactory.createNotification(actionType, task);
notification.sendNotification();
```

Another example of using abstraction is an Observer.java class which is extended by NotificationObserver.java class.

To follow the high cohesion low coupling principles, each of the classes is responsible for concrete functionality. For example NotificationFactory.java is responsible only for creating notifications but not for sending it.

Member 5:

Interfaces were used for abstraction and hierarchy.

So to import and export XML and JSON files **IOController** uses Interface **IParserAdapter**, that later is implemented by classes **XMLParserAdapter** and **JSONParserAdapter**, that are later extended with **XMLCustomParser** and **JSONCustomParser,** instead of implementing all the functionality in one class.

The other example for abstraction are Iterator and TaskCollection that also thave their own interfaces.

High cohesion of the methods was tried to be achieved, by separating tasks and keeping methods simple.

2.5 Defensive Programming

Member 1:
One example of defensive Programming was used in the creation of a Task. For a Task the user should at least provide a Title. To make sure the User provides a Title every time a custom Exception class was used (NoTitleException). This will be thrown after the user clicked on the save button without providing a title. It will then cancel the insertion to the database and show a Toast to the User, that a Title is mandatory.
This is considered defensive programming, because the user cannot do anything to make the app crash while creating a Task.

```
public class NoTitleException extends Exception {
    public NoTitleException(String error){
        super(error);
    }
}
```

```
private boolean checkTitle(String title) throws NoTitleException{
    if (title.trim().isEmpty()) {
        throw new NoTitleException("Title must not be empty.");
    }
    return true;
}
```

```
private void saveTask() {
    String title = editTextTitle.getText().toString();
    String description = editTextDescription.getText().toString();
    location = editTextLocation.getText().toString();

    String textColor = spinnerColor.getSelectedItem().toString();
    String textPriority = spinnerPriority.getSelectedItem().toString();
    String textTaskType = spinnerTaskType.getSelectedItem().toString();

    color = EColor.valueOf(textColor);
    priority = EPriority.valueOf(textPriority);
    taskType = ETaskType.valueOf(textTaskType);

    try{
        checkTitle(title);
    }catch(NoTitleException e){
        Toast.makeText( context: this, e.getMessage(), Toast.LENGTH_SHORT).show();
        return;
    }
```

Log statements were also used for defensive programming. This helps the technical support personnel once the app is deployed and at some point a crash occurs.

Member 3:

To provide correctness and robustness of notification functionality were used defensive programming techniques such as: logs, custom exception classes (NotificationDisabledException.java, UnknownNotificationTypeException.java, UserContactNotFoundException.java)

Example of throwing exceptions to handle cases with invalid data.

```
public Notification createNotification(ActionType actionType, Task task) {

    NotificationSettings notificationSettings =
            notificationSettingsRepository.getByActionType(actionType);

    if (notificationSettings == null)
        throw new UnknownNotificationTypeException("Notification type cannot be null");

    if (!notificationSettings.isEnabled()) {
        throw new NotificationDisabledException();
    }
```

In this example data for sending notifications from the database should be received, if this data was not found in application storage or sending notification is disabled then we throw an exception to enforce a method to stop working. Thrown exceptions might have additional information for further error handling.

Example of logging:

```
try {
    notification = notificationFactory.createNotification(actionType, task);
} catch (UnknownNotificationTypeException e) {
    Log.e( tag: "NotificationService",  msg: "Unknown notification type", e);
    return;
} catch (NotificationDisabledException e) {
    Log.e( tag: "NotificationService",  msg: "Notification disabled", e);
    return;
} catch (UserContactNotFoundException e) {
    Log.e( tag: "NotificationService",  msg: "User contact not found", e);
    return;
}
```

In this example the *createNotification()* method is called, which might produce the exception. Different catch blocks were used for different exceptions to handle them in different ways depending on the exception. In this case none of the exceptions should stop the running program, therefore logging of exception information on error level is used but no further exceptions are thrown.

Member 5:

The key attention was put on correctness and robustness of the program.
That is why main focus was taken on exception handling. And trying to check both input and output values as much as possible.

The main tool for exception handling was try-catch technique.
To collect exceptions more structured we created a class ParsingException that extends class Exception. It is implemented to unify exeption handling.
The class contains exception id and message thrown from exception handling on deeper levels.
So that ParsingException is the only exception thrown by IParserAdapter.

2.6 Testing

Member 1: The responsibility of Member one was tested with instrumented androidTesting
The CRUD Operations where being tried in Unit tests. Here are some snippets of the testing Code:

```
public class DatabaseOperationsTest {

    private AppDatabase appDatabase;
    public TaskDao taskDao;
    public TaskProxy proxy;

    @Before
    public void setup() {
        Context targetContext = InstrumentationRegistry.getInstrumentation().getTargetContext();
        appDatabase =
                Room.inMemoryDatabaseBuilder(targetContext, AppDatabase.class)
                        .allowMainThreadQueries().build();
        taskDao = appDatabase.taskDao();
        proxy = TaskProxy.getInstance(AppDatabase.getDatabase(targetContext));
    }
```

```
@Test
public void insertTaskInDatabase(){

    Appointment input = new Appointment(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Appointment",
            location: "home",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE);
    int outputSizeOfList = 1;

    taskDao.insertAppointment(input);

    List<Appointment> output = taskDao.getAllAppointmentTasks();

    assertEquals(outputSizeOfList, output.size());

}
```

```java
@Test
public void updateTaskInDatabase(){

    Appointment input = new Appointment(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Appointment",
            location: "home",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE);
    String expectedOutput = "test2";

    taskDao.insertAppointment(input);

    List<Appointment> helper = taskDao.getAllAppointmentTasks();
    input = helper.get(0);
    input.setDescription("test2");

    taskDao.updateAppointment(input);

    helper = taskDao.getAllAppointmentTasks();
    Appointment test = helper.get(0);

    String output = test.getDescription();

    assertEquals(expectedOutput, output);

}
```

```
@Test
public void deleteTaskInDatabase(){

    Appointment input = new Appointment(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Appointment",
            location: "home",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE);
    int outputSizeOfList = 0;

    taskDao.insertAppointment(input);

    List<Appointment> output = taskDao.getAllAppointmentTasks();

    input = output.get(0);
    taskDao.deleteAppointment(input);


    output = taskDao.getAllAppointmentTasks();

    assertEquals(outputSizeOfList, output.size());

}
```

```
@Test
public void insertChecklistInDatabase(){

    Checklist input = new Checklist(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Checklist",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE,
            new Time( hour: 13, minute: 0));
    int outputSizeOfList = 1;

    taskDao.insertChecklist(input);

    List<Checklist> output = taskDao.getAllChecklistTasks();

    assertEquals(outputSizeOfList, output.size());
    assertEquals( expected: "Test1", output.get(0).getTitle());

}
```

```java
@Test
public void updateChecklistInDatabase(){

    Checklist input = new Checklist(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Checklist",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE,
            new Time( hour: 13, minute: 0));
    int outputSizeOfList = 1;

    taskDao.insertChecklist(input);

    List<Checklist> output = taskDao.getAllChecklistTasks();

    input = output.get(0);
    input.setDescription("hello");
    taskDao.updateChecklist(input);


    output = taskDao.getAllChecklistTasks();

    assertEquals(outputSizeOfList, output.size());
    assertEquals( expected: "hello", output.get(0).getDescription());

}
```

```
@Test
public void deleteChecklistInDatabase(){

    Checklist input = new Checklist(
            title: "Test1",
            description: "test1",
            EColor.RED,
            EStatus.PLANNED,
            EPriority.LOW,
            subclass: "Checklist",
            new Date( day: 18,  month: 0,  year: 2023),
            ETaskType.PRIVATE,
            new Time( hour: 13, minute: 0));
    int outputSizeOfList = 0;

    taskDao.insertChecklist(input);

    List<Checklist> output = taskDao.getAllChecklistTasks();

    input = output.get(0);
    taskDao.deleteChecklist(input);


    output = taskDao.getAllChecklistTasks();

    assertEquals(outputSizeOfList, output.size());

}
```

PS: some more advanced operations where commented out, because of issues with threads interfering with each other. Before inserting new elements to the database it has to be cleared. But since this runs within a thread the insert operation could have been before, while or after the delete operation. This results in a different result every single time. But constructing the sequence in which the operations were called, it was clear that also the more advanced operations are working fine.

```
@Test
public void insertCorrectTaskClassInDatabase() throws InterruptedException {

    List<Task> toInsert = new ArrayList<>();
    Appointment input = new Appointment("Test1", "test1", EColor.RED, EStatus.PLANNED,EPriority.LOW, "Appointment",
    Checklist input1 = new Checklist("Test2", "test2", EColor.RED, EStatus.PLANNED,  EPriority.LOW,  "Checklist",
    toInsert.add(input);
    toInsert.add(input1);
    int outputSizeOfList = 2;

    proxy.deleteAll();
    for(Task task : toInsert){
        proxy.insert(task);
    }

    List<Appointment> appointments = proxy.getAllAppointmentTasks();
    List<Checklist> checklists = proxy.getAllChecklistTasks();

    List<Task> output = new ArrayList<>();
    output.addAll(appointments);
    output.addAll(checklists);

    for(Appointment item : appointments){
        Log.d(TAG, "inside appointments we have uids: " + item.getUid() + ", with title: " + item.getTitle());
    }
    for(Checklist item : checklists){
        Log.d(TAG, "inside checklists we have uids: " + item.getUid() + ", with title: " + item.getTitle());
    }

    assertEquals(1, appointments.size());
    assertEquals(1, checklists.size());
    assertEquals(outputSizeOfList, output.size());
    assertEquals("Appointment", output.get(0).getSubclass());
    assertEquals("Checklist", output.get(1).getSubclass());
```

Member 3:
For testing notifications related functionality were used unit and manual testing. With unit tests all of the related CRUD operations in Dao and Repository classes were covered. Tests are implemented in the following classes:

- NotificationSettingsDaoTest.java
- NotificationSettingsRepositoryTest.java
- UserContactDaoTest.java
- UserContactRepositoryTest.java

In addition, the NotificationFactory class is also covered with tests, as it is the main class for creating notifications and defining which type of the notification should be created. Tests are implemented in the following class: NotificationFactoryTest.java .

Since sending notification functionality can go beyond the application (email sending) it was covered by manual testing using https://mailtrap.io service.

Also, manual testing covered all of the interactions with an application through UI.

Member 5:

The whole main functionality and the most probable errors of the import/export of JSON and XML files were tested.

For testing purposes 3 test classes and 6 data files were created.

Test JSON and XML files are saved under the path: implementation/app/example.

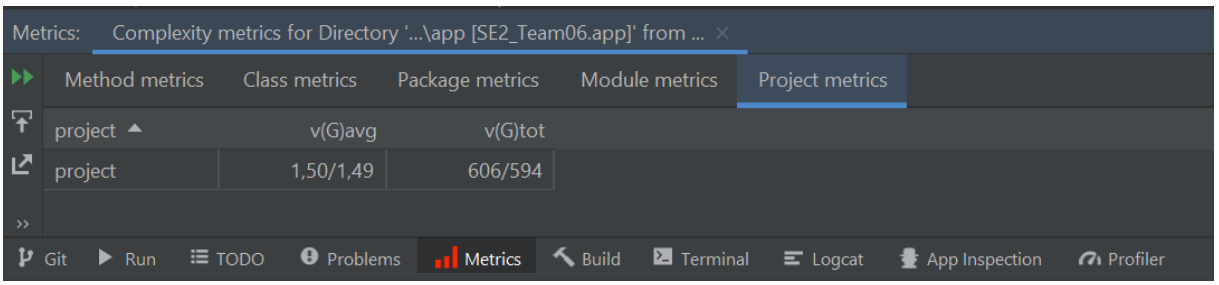
1. JSONCustomParserUnitTest class is used to check functionality of JSON format parsing. It contains the following test types:
    - Exporting Appointment and Checklist objects with different values
    - Importing/parsing Strings and saving them as tasks
    - Testing different parsing exceptions, such as type mismatch, no attribute, no opening element, empty file
    - Check equals methods for TaskCollections

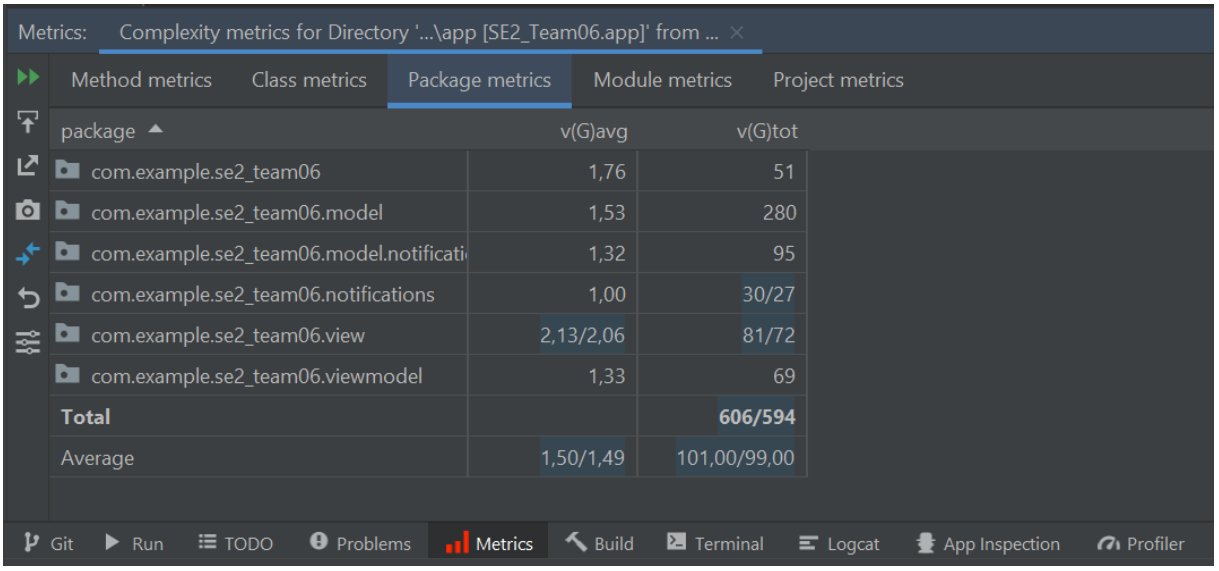
2. XMLCustomParserUnitTest class is used to check functionality of JSON format parsing. It contains the following test types:
    - Exporting Appointment and Checklist objects with different values
    - Importing/parsing Strings and saving them as tasks
    - Testing different parsing exceptions, such as type mismatch, no attribute, no opening element, empty file

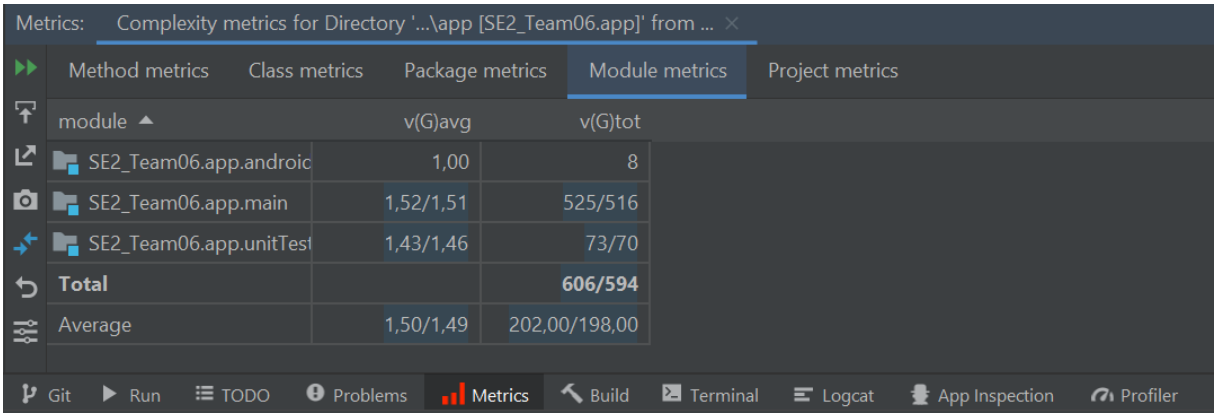
3. TaskCollectionTest class was used to test TaskCollection to check, if the class saves tasks properly.

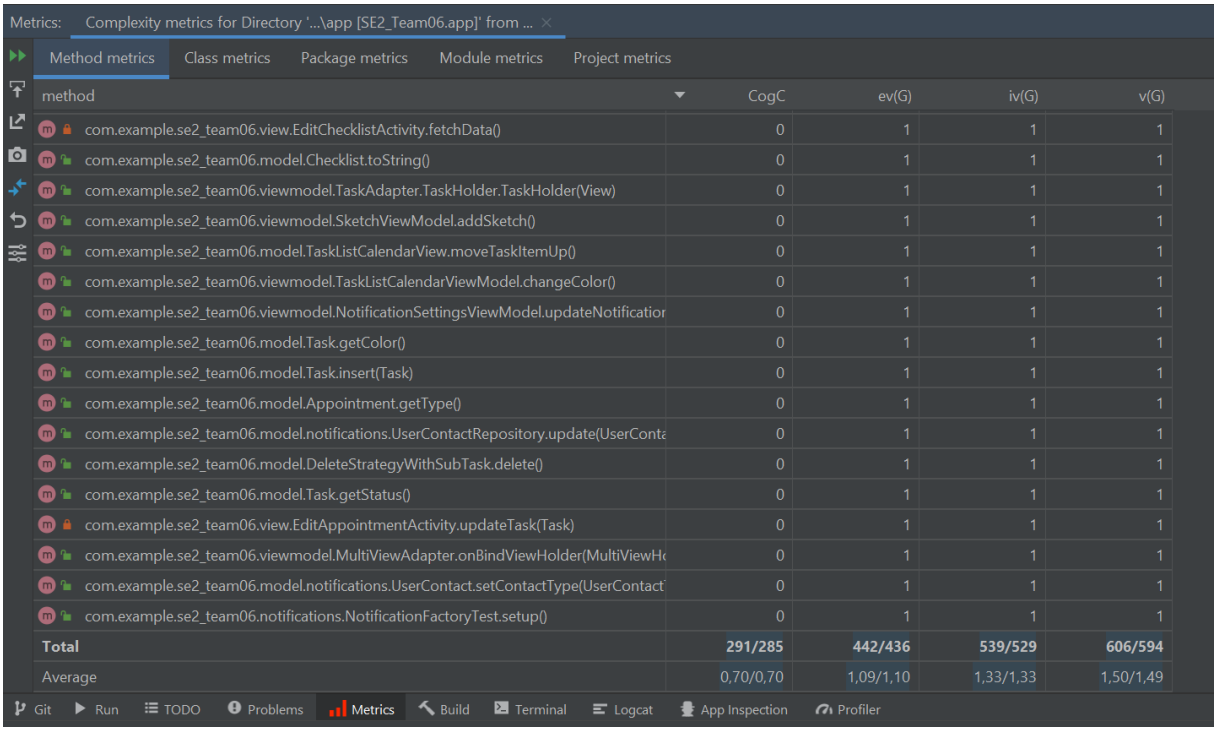
3 Code Metrics

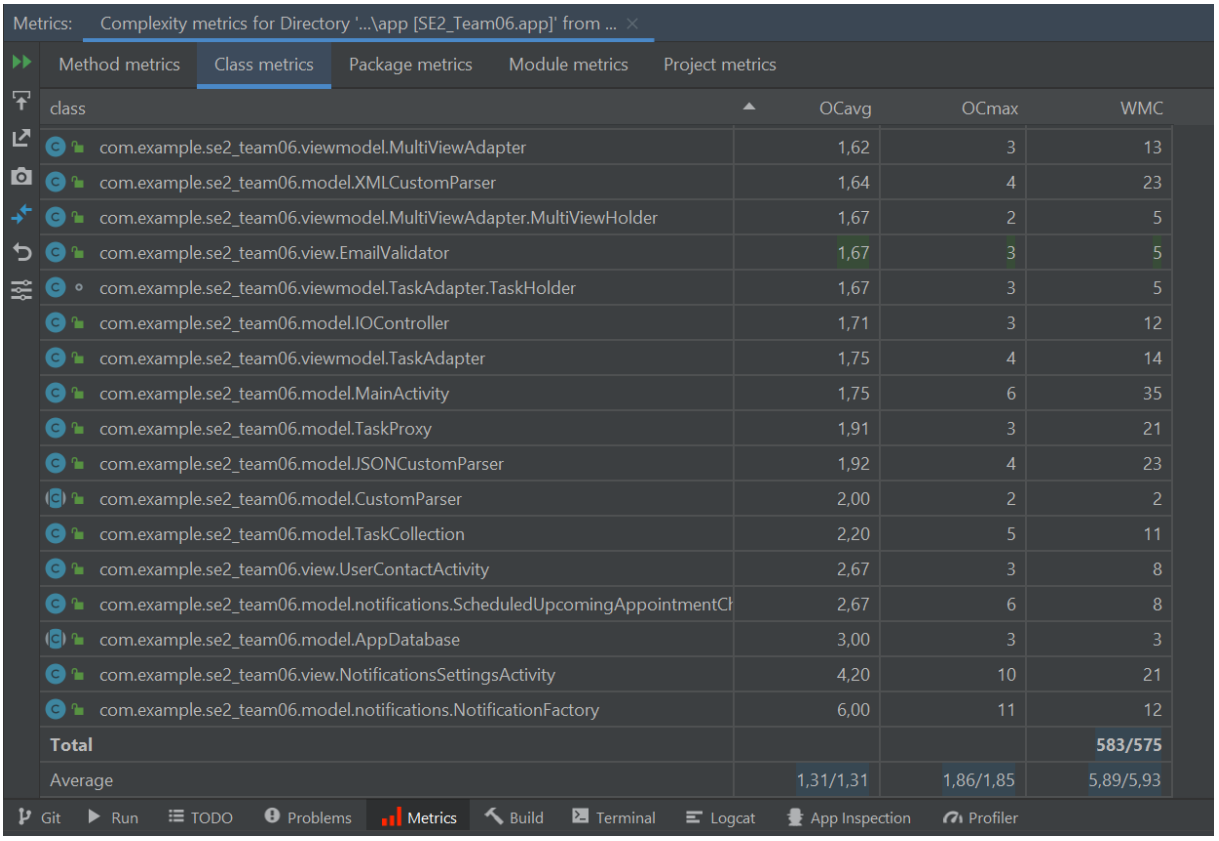
Here are the the metrics findings of the current application:




Metrics: Complexity metrics for Directory '...\app [SE2_Team06.app]' from ...
Method metrics
Class metrics
Package metrics
Module metrics
Project metrics
package
v(G)avg
v(G)tot
com.example.se2_team06 1,76 51
com.example.se2_team06.model 1,53 280
com.example.se2_team06.model.notificati 1,32 95
com.example.se2_team06.notifications 1,00 30/27
com.example.se2_team06.view 2,13/2,06 81/72
com.example.se2_team06.viewmodel 1,33 69
Total 606/594
Average 1,50/1,49 101,00/99,00
Git
Run
TODO
Problems
Metrics
Build
Terminal
Logcat
App Inspection
Profiler


Metrics: Complexity metrics for Directory '...\app [SE2_Team06.app]' from ...
Method metrics
Class metrics
Package metrics
Module metrics
Project metrics
module
v(G)avg
v(G)tot
SE2_Team06.app.androic 1,00 8
SE2_Team06.app.main 1,52/1,51 525/516
SE2_Team06.app.unitTest 1,43/1,46 73/70
Total 606/594
Average 1,50/1,49 202,00/198,00
Git
Run
TODO
Problems
Metrics
Build
Terminal
Logcat
App Inspection
Profiler


Metrics: Complexity metrics for Directory '...\app [SE2_Team06.app]' from ...
Method metrics
Class metrics
Package metrics
Module metrics
Project metrics
method
CogC
ev(G)
iv(G)
v(G)
com.example.se2_team06.view.EditChecklistActivity.fetchData() 0 1 1 1
com.example.se2_team06.model.Checklist.toString() 0 1 1 1
com.example.se2_team06.viewmodel.TaskAdapter.TaskHolder.TaskHolder(View) 0 1 1 1
com.example.se2_team06.viewmodel.SketchViewModel.addSketch() 0 1 1 1
com.example.se2_team06.model.TaskListCalendarView.moveTaskItemUp() 0 1 1 1
com.example.se2_team06.viewmodel.TaskListCalendarViewModel.changeColor() 0 1 1 1
com.example.se2_team06.viewmodel.NotificationSettingsViewModel.updateNotificatior 0 1 1 1
com.example.se2_team06.model.Task.getColor() 0 1 1 1
com.example.se2_team06.model.Task.insert(Task) 0 1 1 1
com.example.se2_team06.model.Appointment.getType() 0 1 1 1
com.example.se2_team06.model.notifications.UserContactRepository.update(UserConta 0 1 1 1
com.example.se2_team06.model.DeleteStrategyWithSubTask.delete() 0 1 1 1
com.example.se2_team06.model.Task.getStatus() 0 1 1 1
com.example.se2_team06.view.EditAppointmentActivity.updateTask(Task) 0 1 1 1
com.example.se2_team06.viewmodel.MultiViewAdapter.onBindViewHolder(MultiViewH 0 1 1 1
com.example.se2_team06.model.notifications.UserContact.setContactType(UserContact 0 1 1 1
com.example.se2_team06.notifications.NotificationFactoryTest.setup() 0 1 1 1
Total 291/285 442/436 539/529 606/594
Average 0,70/0,70 1,09/1,10 1,33/1,33 1,50/1,49
Git
Run
TODO
Problems
Metrics
Build
Terminal
Logcat
App Inspection
Profiler

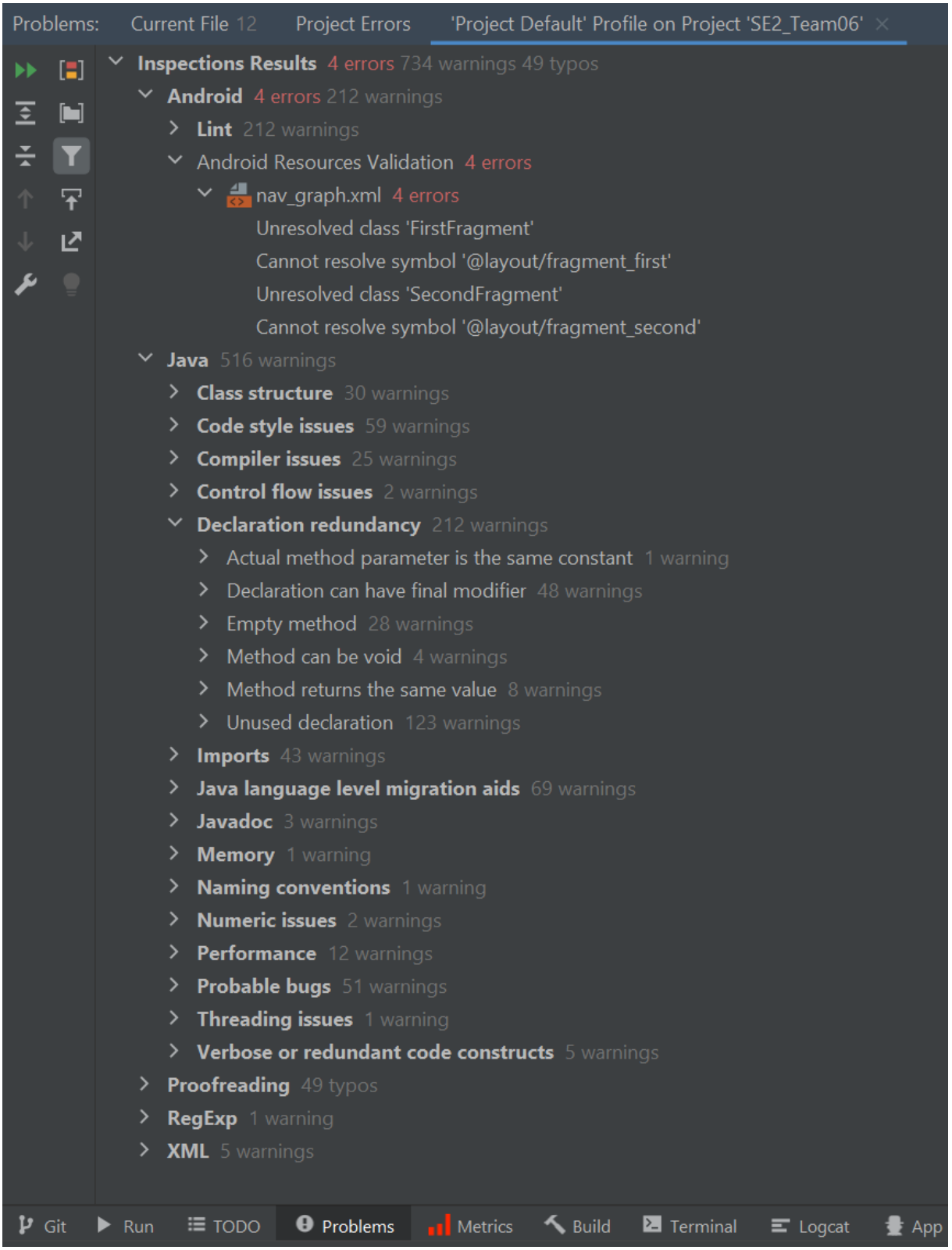


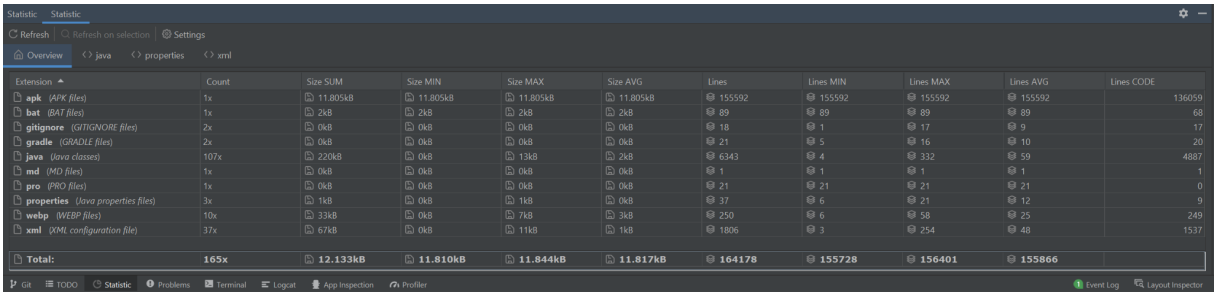
Metrics: Complexity metrics for Directory '...\app [SE2_Team06.app]' from ...

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

class	OCavg	OCmax	WMC
com.example.se2_team06.viewmodel.MultiViewAdapter	1,62	3	13
com.example.se2_team06.model.XMLCustomParser	1,64	4	23
com.example.se2_team06.viewmodel.MultiViewAdapter.MultiViewHolder	1,67	2	5
com.example.se2_team06.view.EmailValidator	1,67	3	5
com.example.se2_team06.viewmodel.TaskAdapter.TaskHolder	1,67	3	5
com.example.se2_team06.model.IOController	1,71	3	12
com.example.se2_team06.viewmodel.TaskAdapter	1,75	4	14
com.example.se2_team06.model.MainActivity	1,75	6	35
com.example.se2_team06.model.TaskProxy	1,91	3	21
com.example.se2_team06.model.JSONCustomParser	1,92	4	23
com.example.se2_team06.model.CustomParser	2,00	2	2
com.example.se2_team06.model.TaskCollection	2,20	5	11
com.example.se2_team06.view.UserContactActivity	2,67	3	8
com.example.se2_team06.model.notifications.ScheduledUpcomingAppointmentCl	2,67	6	8
com.example.se2_team06.model.AppDatabase	3,00	3	3
com.example.se2_team06.view.NotificationsSettingsActivity	4,20	10	21
com.example.se2_team06.model.notifications.NotificationFactory	6,00	11	12
Total			**583/575**
Average	1,31/1,31	1,86/1,85	5,89/5,93

Git | Run | TODO | Problems | Metrics | Build | Terminal | Logcat | App Inspection | Profiler


Problems: Current File 12 Project Errors 'Project Default' Profile on Project 'SE2_Team06'
Inspections Results 4 errors 734 warnings 49 typos
Android 4 errors 212 warnings
Lint 212 warnings
Android Resources Validation 4 errors
nav_graph.xml 4 errors
Unresolved class 'FirstFragment'
Cannot resolve symbol '@layout/fragment_first'
Unresolved class 'SecondFragment'
Cannot resolve symbol '@layout/fragment_second'
Java 516 warnings
Class structure 30 warnings
Code style issues 59 warnings
Compiler issues 25 warnings
Control flow issues 2 warnings
Declaration redundancy 212 warnings
Actual method parameter is the same constant 1 warning
Declaration can have final modifier 48 warnings
Empty method 28 warnings
Method can be void 4 warnings
Method returns the same value 8 warnings
Unused declaration 123 warnings
Imports 43 warnings
Java language level migration aids 69 warnings
Javadoc 3 warnings
Memory 1 warning
Naming conventions 1 warning
Numeric issues 2 warnings
Performance 12 warnings
Probable bugs 51 warnings
Threading issues 1 warning
Verbose or redundant code constructs 5 warnings
Proofreading 49 typos
RegExp 1 warning
XML 5 warnings
Git Run TODO Problems Metrics Build Terminal Logcat App



As we can clearly see, the number of lines of code have risen, as expected. Now let's compare it to the beginning of this project

	Old Metrics	Current Metrics
Java classes	79	107
Java lines of code	1462	6343
xml files	25	37
xml lines of code	773	1806
Problems	4	4 (the same but have nothing to do with our implementation)

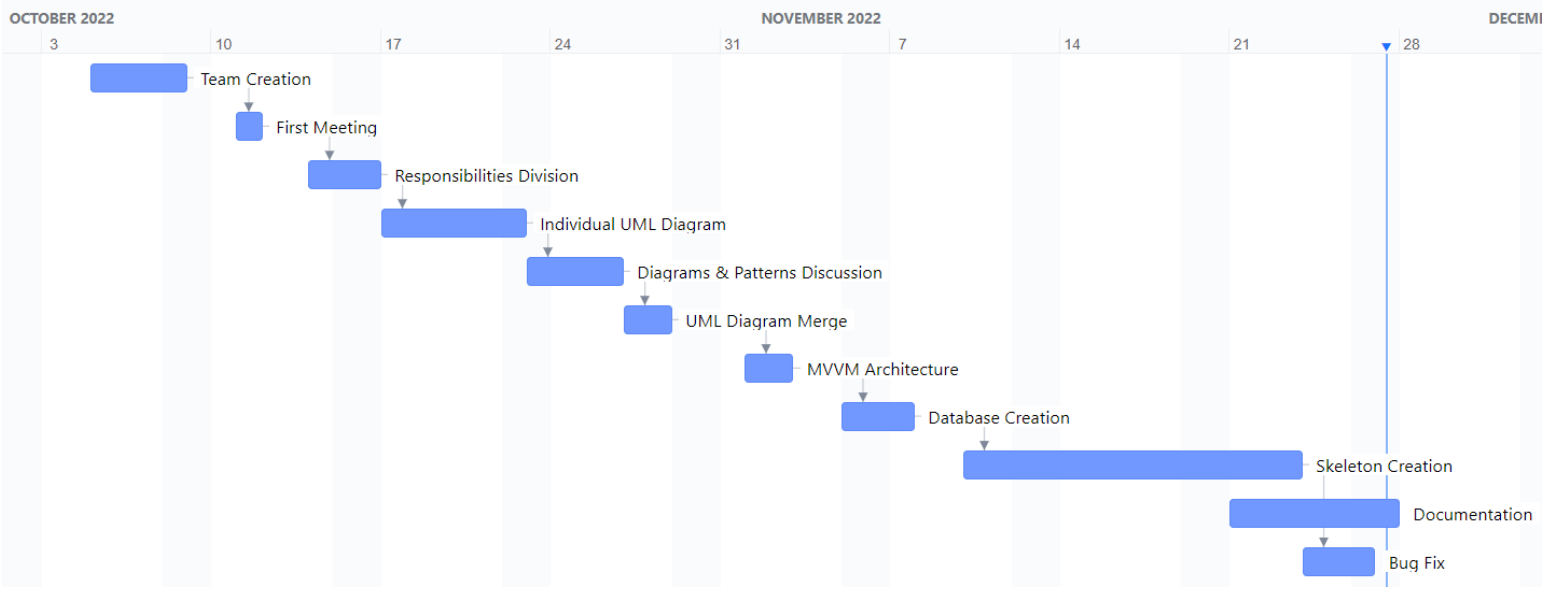
The results are very pleasing, thus the java classes and xml files did not multiply too much. This means that our initial approach was very well thought through and we considered much that is going to be needed for the project.

4 Team Contribution

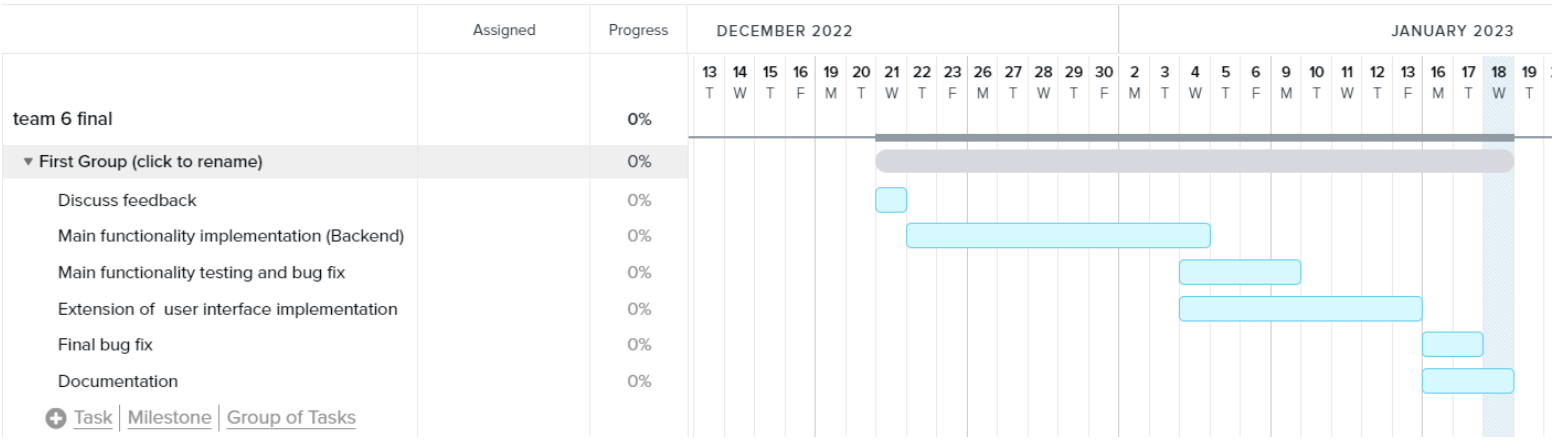
4.1 Project Tasks and Schedule

DESIGN

TITLE	DURATION	START / END
Team Creation	3	Oct 05 - Oct 08
First Meeting	1	Oct 11 - Oct 11
Responsibilities Division	1	Oct 14 - Oct 16
Individual UML Diagram	5	Oct 17 - Oct 22
Diagrams & Patterns Discussion	3	Oct 23 - Oct 26
UML Diagram Merge	2	Oct 27 - Oct 28
MVVM Architecture	2	Nov 01 - Nov 02
Database Creation	1	Nov 05 - Nov 07
Skeleton Creation	10	Nov 10 - Nov 23
Documentation	5	Nov 21 - Nov 27
Bug Fix	2	Nov 24 - Nov 26



FINAL



4.2 Distribution of Work and Efforts

Contribution of member 1:

Member 1 is responsible for the CRUD Operation with 2 Task-Types. These stand for Creation (and saving in database), Reading from database, Updating in the database and deleting from the database. The basic foundation of this responsibility was implemented in the first go, which made the progress a lot faster. Though it was tricky to make it work with another Subclass and have it visually representing (which I did since member 4 dropped out). I implemented the Proxy pattern as a level of indirectness to execute the CRUD operations in a safe and correct manner. The Factory pattern was used for the Creation part, where an instance of a Subclass was assigned to an instance of the super class. Like that the Proxy could handle the insertion with a parameter Task, by checking which class lies underneath. Thich makes future subclasses easily integratable. Creating Views for every single CRUD and Subclass was time consuming. Also the Views for the two different RecyclerView with the items and Adapters.

Creation of tasks: 5h
Designing Patterns: 10h
Debugging: 20h
Adapter view: 5h
Reading from Database: 5h
Updating of tasks: 4h
Deletion of tasks: 3h
Views for CRUD: 10h
multi view & multi select: 10h
Recycler View: 5h

Writing documentation: 3h

Contribution of member 3:

DESIGN

1. Documentation of application components for database, class diagram overview, technology stack, individual parts for design patterns description. 5 hours
2. Design patterns: Decorator, Observer, Factory (for notification functionality). 15 hours
3. Skeleton of implementation of notifications related logic with using design patterns. 10 hours
4. Set up a Room database. 5 hours
5. Created basic UI for notification and account settings. 3 hours
6. Design UML diagram. 10 hours

FINAL

1. Adaptation of UML diagram. 2 hours
2. Implementation of design patterns Decorator, Observer, Factory. 10 hours
3. Creating a component diagram. 1 hour
4. Implementation of sending pop-up and email notifications. 15 hours
5. Extending UI for User Contact and Notification Settings. 5 hours
6. Extending Model and ViewModel classes. 10 hours
7. Manual testing. 7 hours
8. Covering CRUD operations with unit testing. 7 hours
9. Debugging. 7 hours
10. Documentation. 3 hours

Contribution of member 5:

DESIGN

1. Create UML diagram. 20 hours
2. Implement design patterns: Adapter Method, Iterator Pattern. 10 hours
3. Skeleton for import/export of tasks from/to JSON and XML files. 30 hours
4. Add dependency org.json.simple to gradle. 1 hour
5. Fix bugs. 5 hours
6. Documentation of individual parts, design approach and overview, project tasks and schedule sections. 5 hours

FINAL

1. Update UML diagram and create UML component diagram. 1 hour
2. Implementation of main import/export methods. 20 hours
3. Implementation of IOController. 7 hours
4. Implement UserInterface. 5 hours

3. Testing. 5 hours
4. Fix bugs. 5 hours
5. Documentation. 3 hours

4.3 How-To Documentation

Link to the video: https://youtu.be/p6Iw6A6Pd20

To install APK file you need:
1) Download app-debug.apk from gitlab
2) Open Android Studio
3) Select File -> Profile or Debug APK
4) Select downloaded apk file
5) After installing you can run the application

Or another way:
To install the APK on your Android Emulator, you simply drag and drop the APK to the Emulator and it will be automatically installed. After it has been installed successfully, you find the icon with the name “SE2_team0206” and click it.
The App is now open.

To use the Task Manager one can click the right bottom button to start creating a Task (Appointment or Checklist). Once decided, one will be directed to the Creation View where you can set parameters like Title, description, Color and so on. After you are done you can click the Save Button on the bottom to create this Task and save it in the database.

After that you will come back to the first View and will see your now freshly created Task. By clicking on this item you will be redirected to the Update View where you see all the information the Task and can change them and Update them, or even delete it.

The item Checklist differs from the item Appointment on the List View, that you have a check box for the Checklist where you can tick off your Task.

Clicking on the top right Button Select you will now be able to select multiple Tasks in your list and can either update them or delete them. For updating you have the option to set a new Priority for all the selected items.

For testing Email functionality we created an account in Mailtrap you need to login to check the inbox.

Credentials for Email notification sending platform https://mailtrap.io:
Email : team0202se2@yahoo.com
Password: se22022W

Yahoo account:
Email: team0202se2@yahoo.com
Password: HelloWorld2022W