

Software Engineering 2

DESIGN 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 Design Draft

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 (read) the task list that is to be exported or imported from the database.

Application components for database

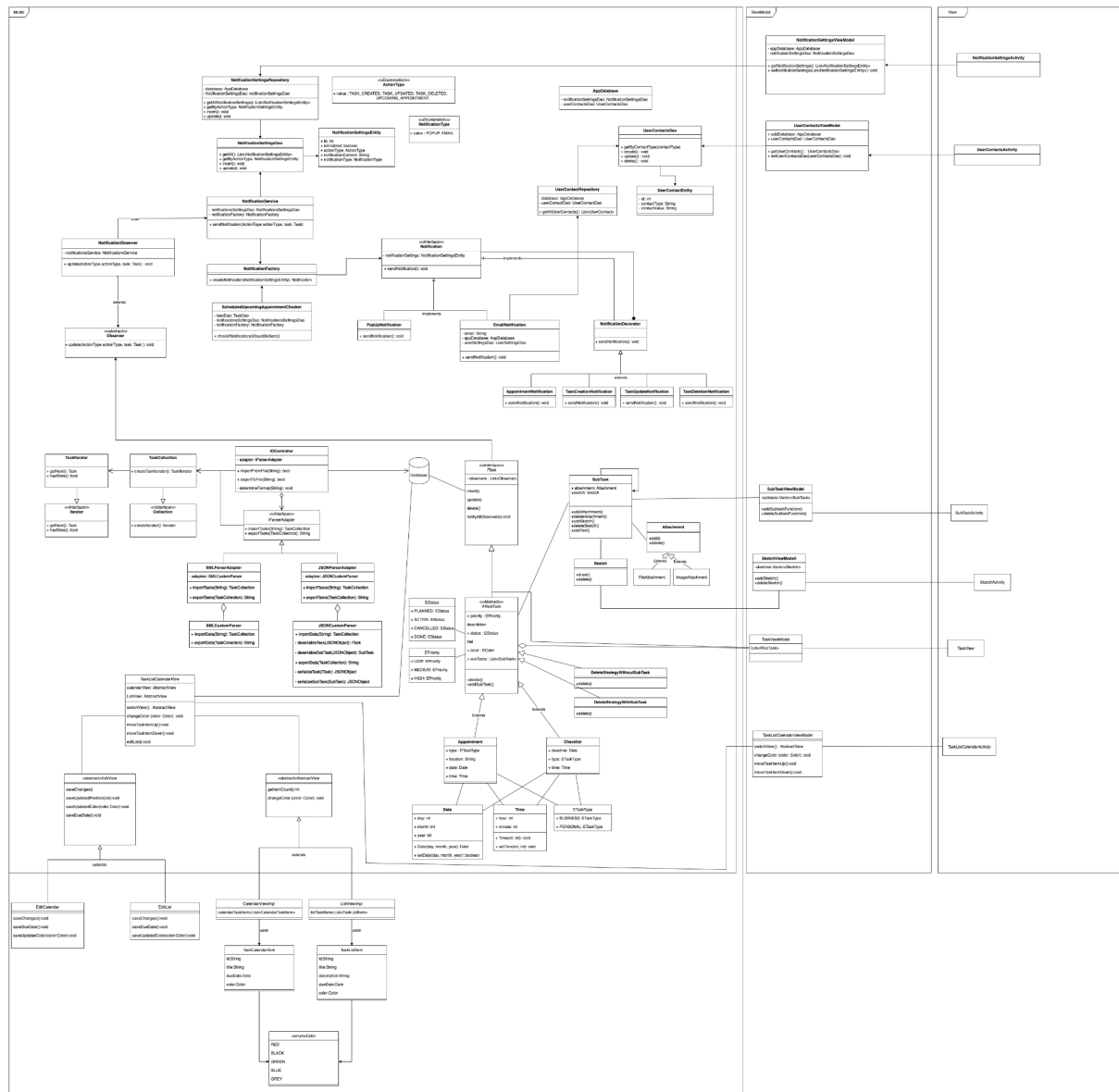
Since we decided to use the Room persistence library for SQLite, we created a database design architecture according to the guidelines from the site <https://developer.android.com>. Primary database design components are: database class, Data entity classes, Data Access Objects (DAOs) classes and repository 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:

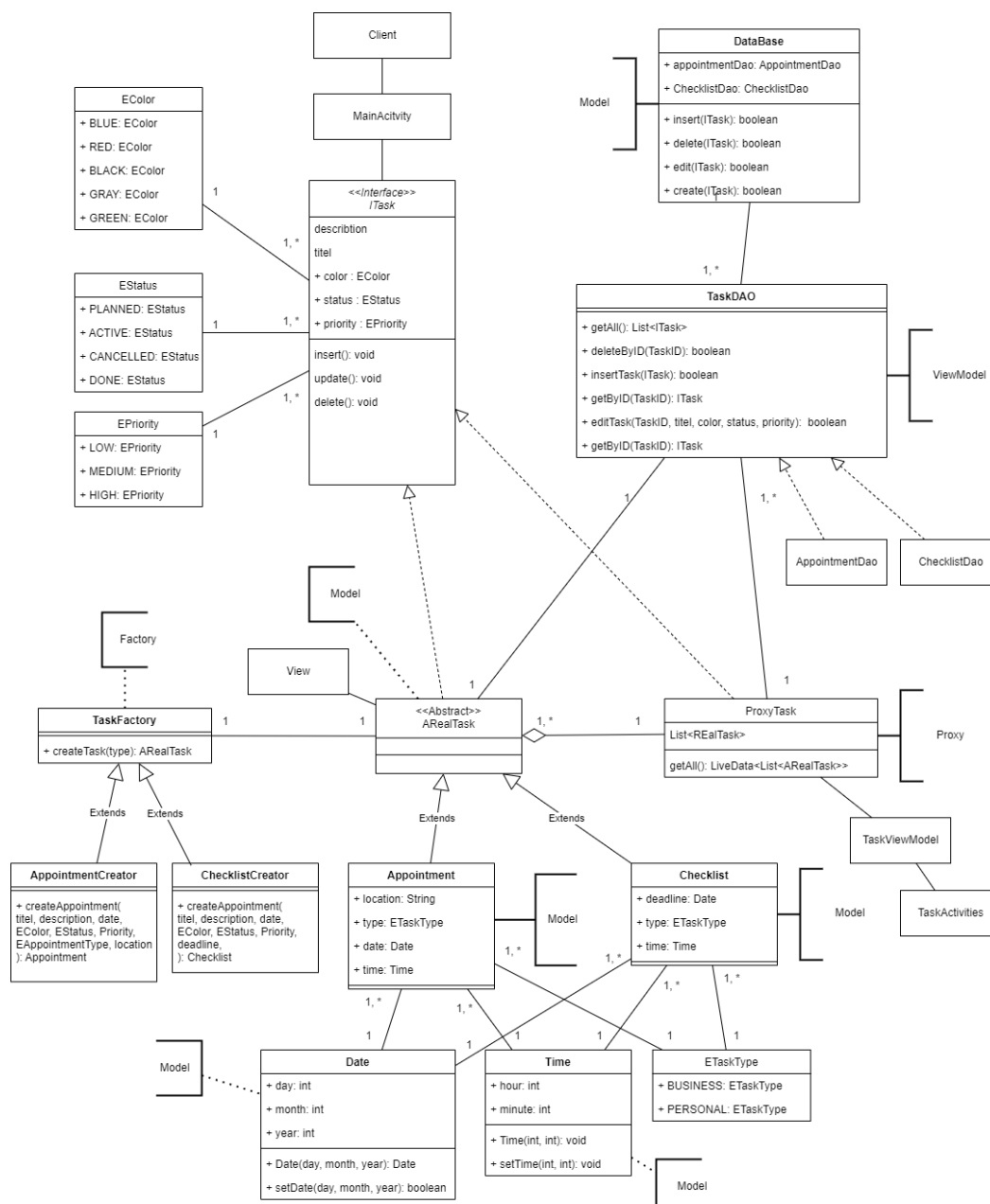
- Abstract class *AppDatabase* which extends *RoomDatabase* and defines all database configurations. We are using this class as the main entry point to our persisted data.
- Entity classes: *Time*, *Date*, *Checklist*, *Appointment*, *Task*, *NotificationSettings*, *UserContact*.
- DAO classes: *AppointmentDao*, *ChecklistDao*, *NotificationSettingsDao*, *UserContactDao*
- 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

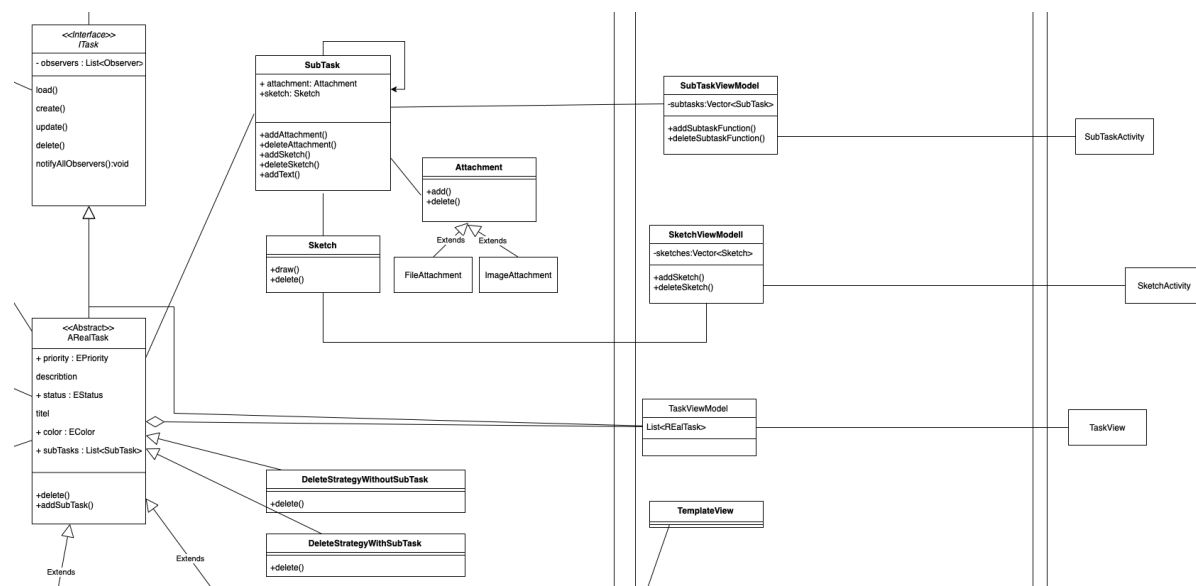
The Member 2 responsibility was to implement the suitable strategy for deletion of parent and/or subtasks, to ensure users can create a sub-task and to make hand written notes. To ensure that each task type supports at least two file format attachments.

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 SketchActivity 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).

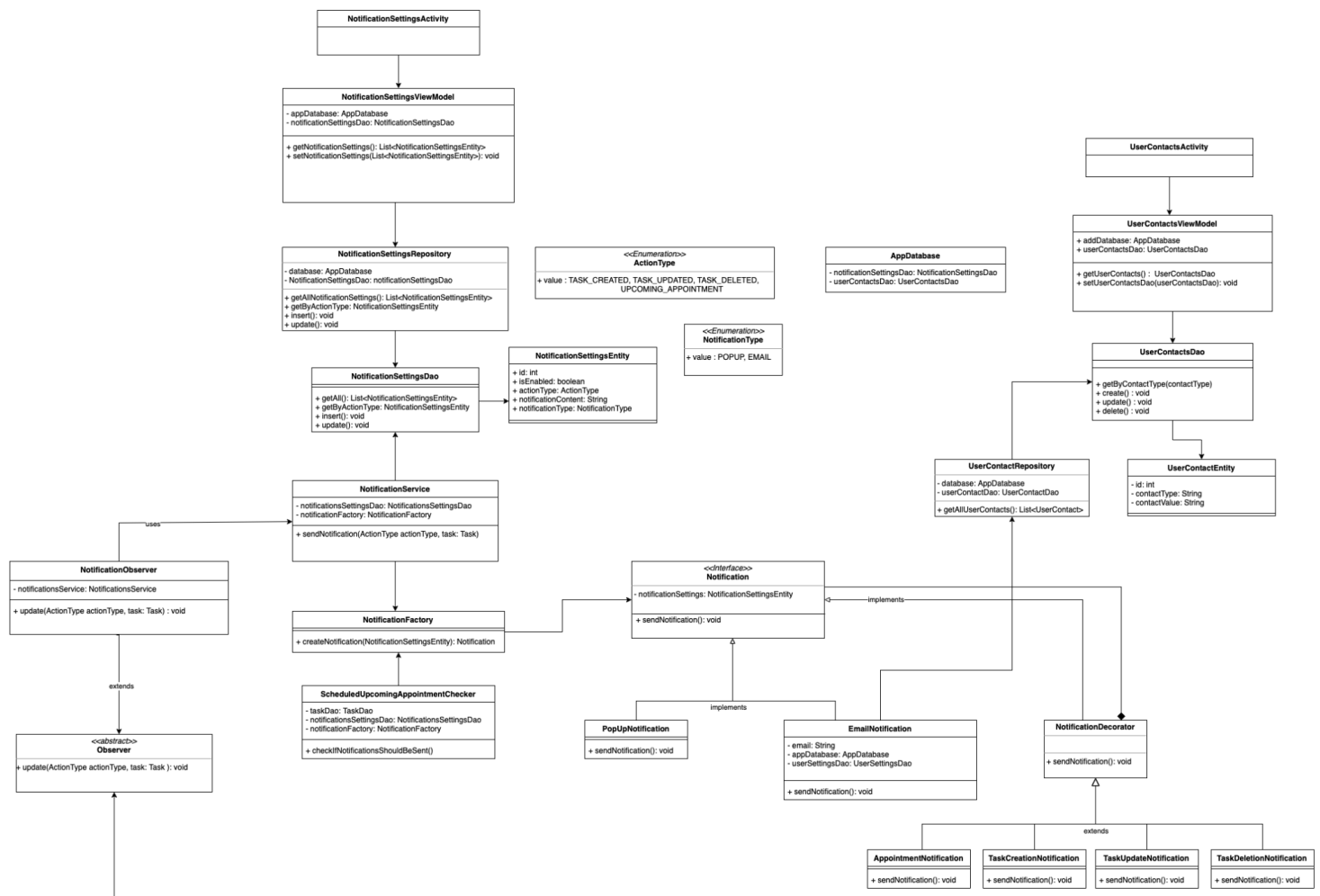
All of classes are structured by layers:

- View layer for notifications functionality consists of *NotificationSettingsActivity* and *UserContactActivity* classes.
- ViewModel layer classes are: *NotificationSettingsViewModel* and *UserContactViewModel*.
- Model layer components are: *NotificationSettingsEntity*, *UserContactEntity*, *NotificationSettingsDao*, *UserContactDao*, *NotificationSettingsRepository*, *UserContactRepository*, *NotificationType*, *ActionType*, *NotificationService*, *NotificationFactory*, *ScheduledUpcomingAppointmentChecker*, *Notification*, *PopUpNotification*, *EmailNotification*, *NotificationDecorator*, *AppointmentNotification*, *TaskCreationNotification*, *TaskUpdateNotification*, *TaskDeletionNotification*, *Observer*, *NotificationObserver*.

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

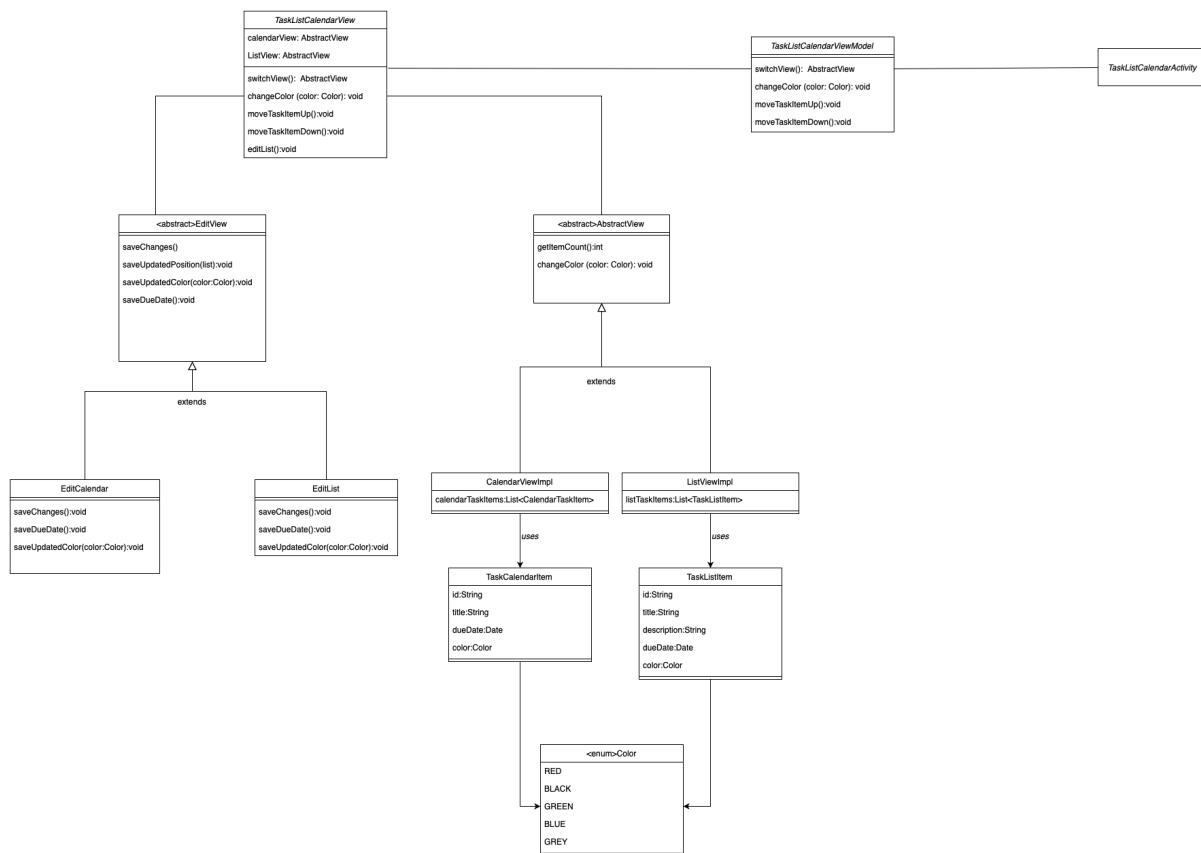
The application supports customizable views, such as list view and calendar view. Also it should be possible for the user to reorder tasks in a list view and hide or show chosen tasks.

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.

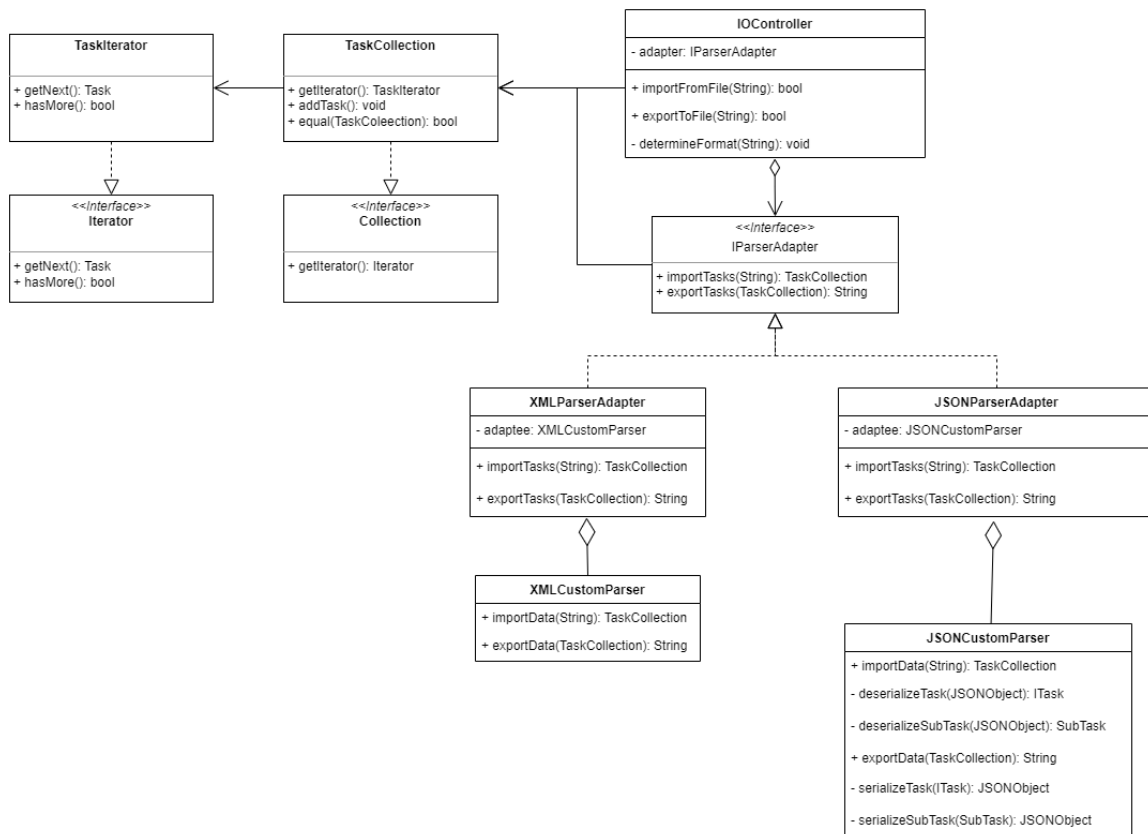
Model: TaskListCalendarView responsible for saving in DB chosen colour for the task, the updated task position and chosen view.



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 architecture. The diagram will probably be extended in the future with classes in the View part, as we will need a separate 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

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.

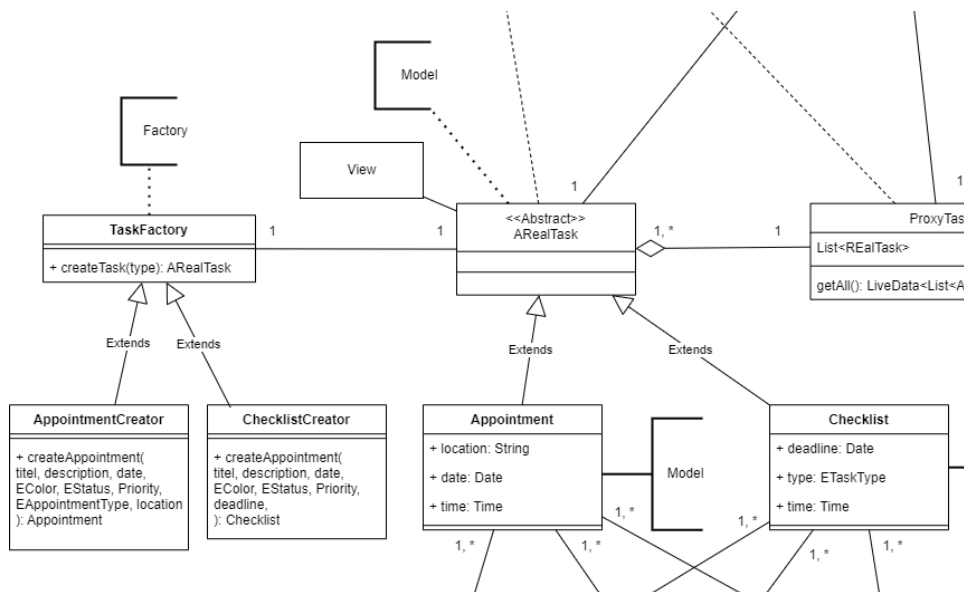
According to the project developing recommendations, we use API level 28 of Android 9.0 Pie and Phone Pixel 2 as Android virtual device.

1.2 Design Patterns

1.2.1 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.



```
private Task toBeInserted;
```

```
switch (subclass) {
    case "Appointment":
        toBeInserted = new Appointment(title, description, EColor.RED, EStatus.ACTIVE, EPriority.MEDIUM, location: "location", new Date( day: 1, month: 1, year: 2022), ETaskType.PRIVATE);
        taskViewModel.insert(toBeInserted);
        break;
    case "Checklist":
        //toBeInserted = new Checklist(title, description, color, status, priority, deadline, type, time);
        //taskViewModel.insert(toBeInserted);
        break;
    default:
        Toast.makeText( context: MainActivity.this, text: "Task is not valid", Toast.LENGTH_SHORT).show();
}
}
```

Factory Pattern for Notification functionality:

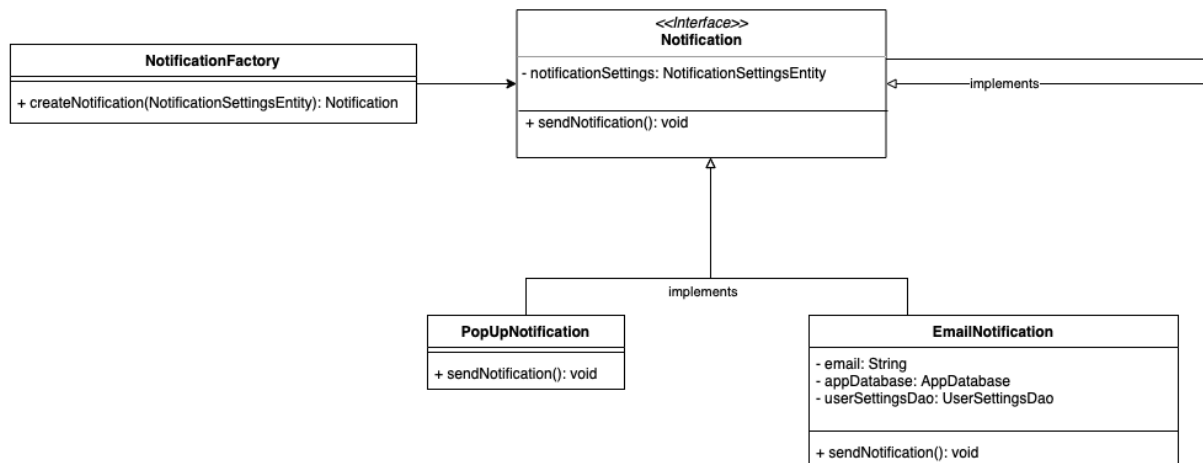
The Factory Pattern is the approach which helps us to flexibly create new objects without directly instantiating these objects.

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.



```
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");  
        }  
    }  
}
```

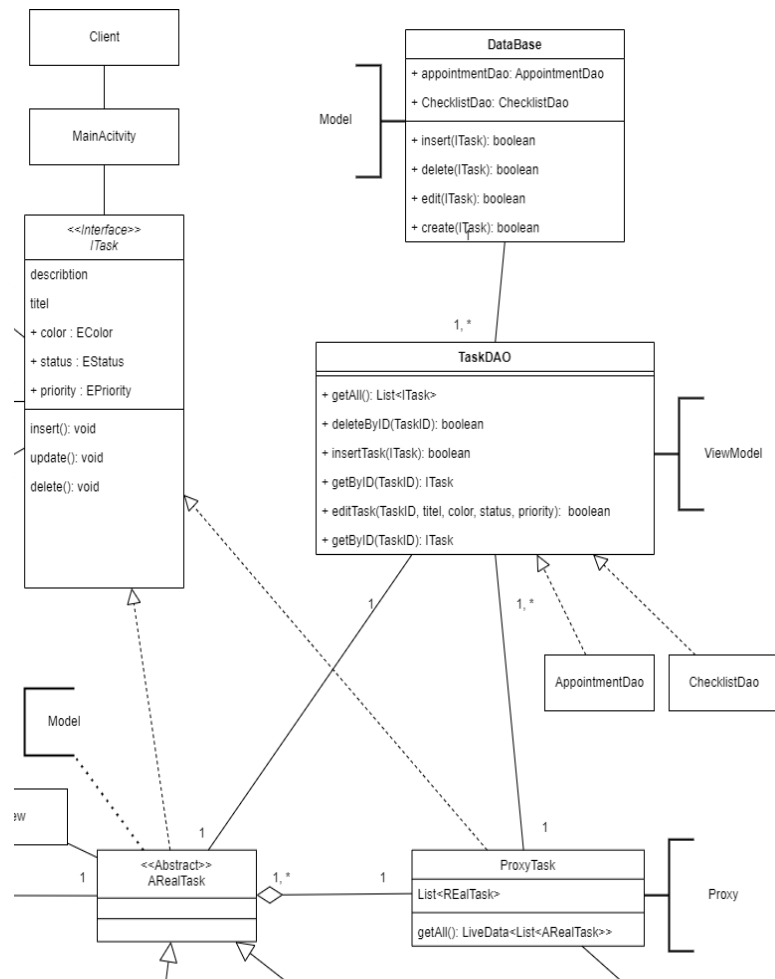
```
public class PopUpNotification implements Notification{  
    @Override  
    public void sendNotification(){}  
}
```

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

1.2.2 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 AppointmentDao appointmentDao;
    private ChecklistDao checklistDao;
    private LiveData<List<Task>> allTasks;

    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;
        appointmentDao = database.appointmentDao();
        checklistDao = database.checklistDao();
        allTasks = appointmentDao.getAll();
    }
}
```



```
public LiveData<List<Task>> getAll() { return allTasks; }

@Override
public void insert(Task task) {
    if(task instanceof Appointment) {
        Appointment appointment = (Appointment) task;
        AppDatabase.databaseWriteExecutor.execute(() -> {
            appointmentDao.insertAll(appointment);
            Log.d(TAG, msg: "insert() returned: " + "new task saved");
        });
    }
}

@Override
public void update(Task task) {

}

@Override
public void delete(Task task) {

}

public void deleteAll(){

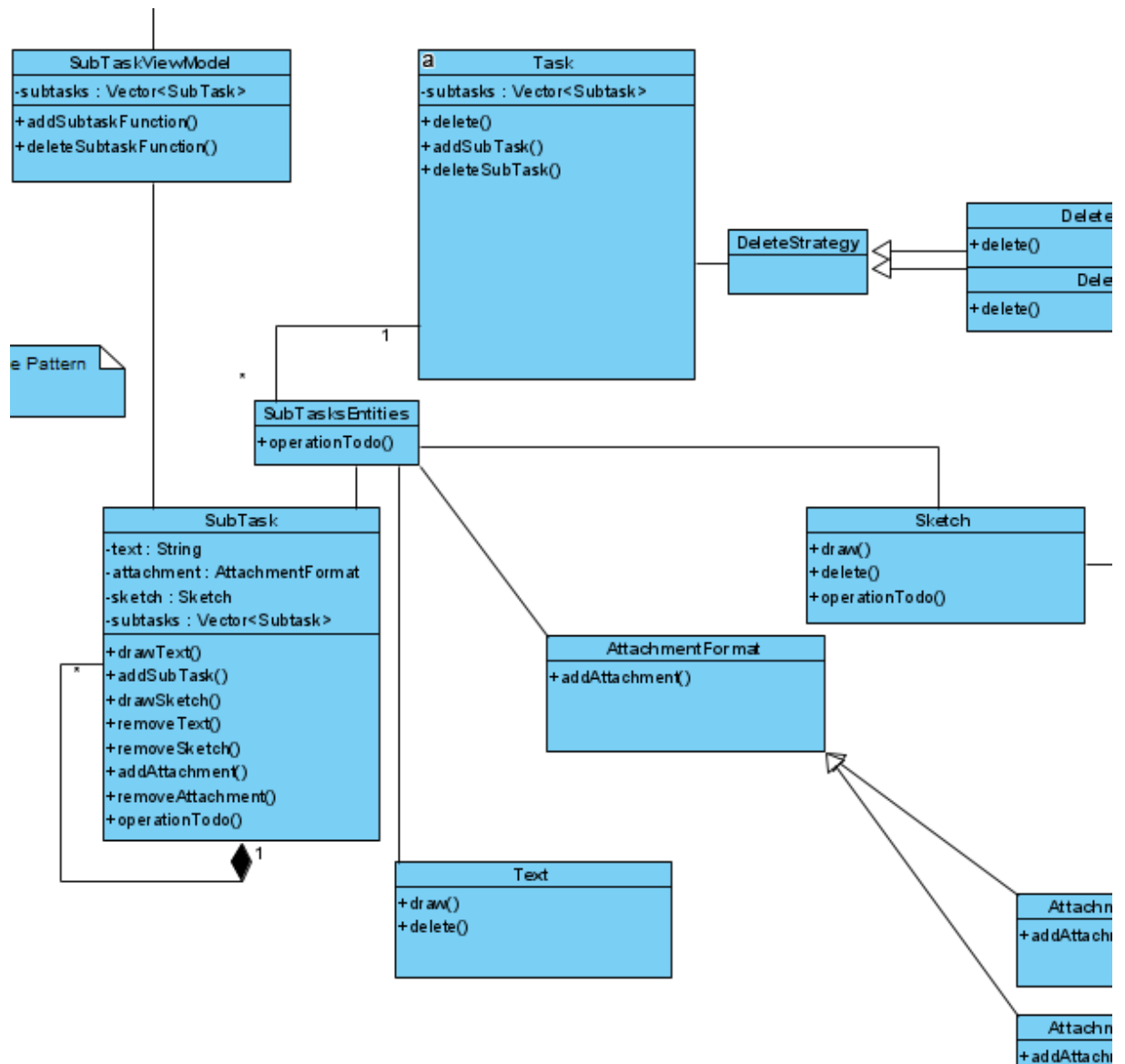
}

@Override
public void notifyAllObservers() {

}
```

1.2.3 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.



•

```

package com.example.se2_team06.model;

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

Interface

```
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() {

    }
}
```

Leaf1

```

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() {

    }

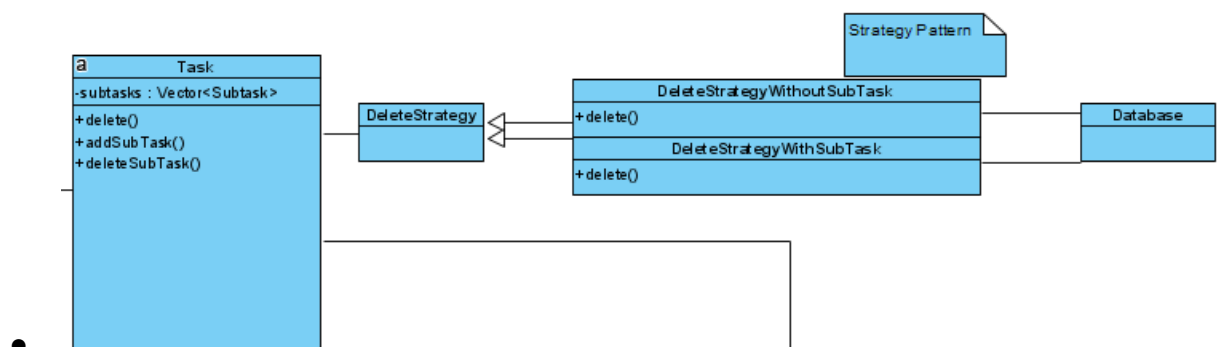
}

```

Leaf2

1.2.4 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.

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

-

```
package com.example.se2_team06.model;  
  
public class DeleteStrategyWithoutSubTask implements DeleteStrategy{  
    @Override  
    public void delete() {  
        //Delete without subtasks  
    }  
}
```

```
package com.example.se2_team06.model;  
  
public class DeleteStrategyWithSubTask implements DeleteStrategy{  
    @Override  
    public void delete() {  
        //Delete with subtasks  
    }  
}
```

1.2.5 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.

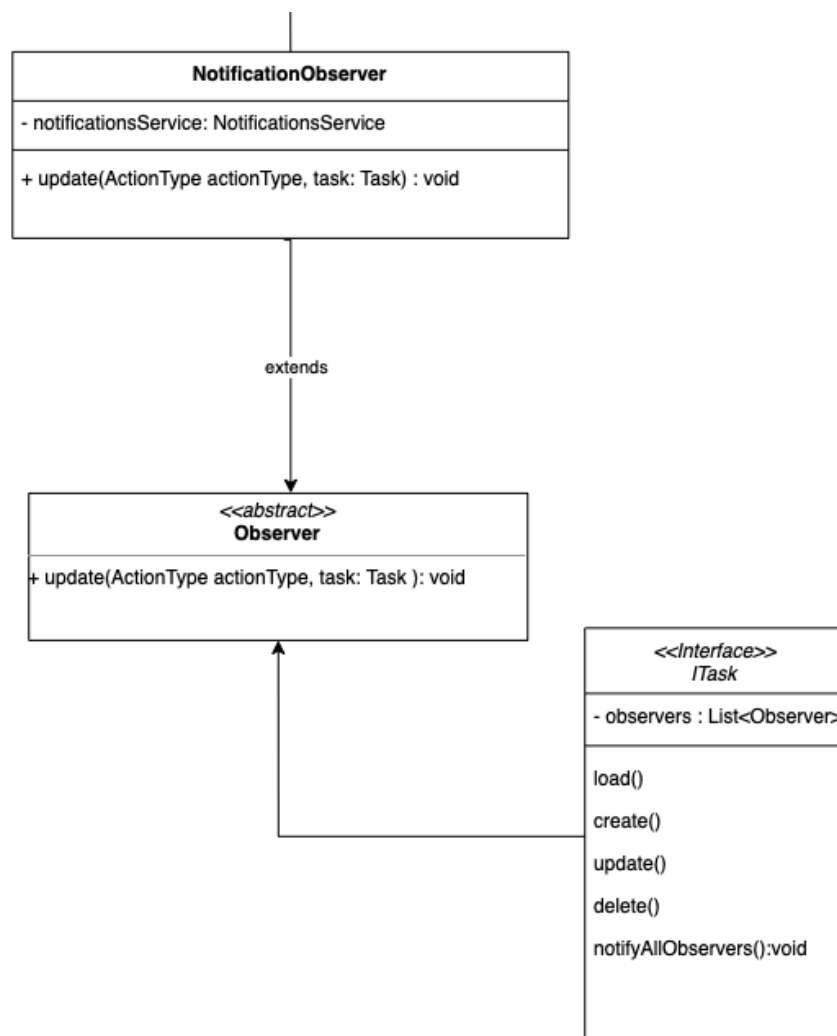
The problem, that the Observer pattern solves, is that the notification service doesn't need to check the state of the task to send a notification, instead everything proceeds automatically. And also to prevent the user from not being notified of any action.

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.



```
public interface ITask {  
    List<Observer> observers = null;  
  
    void insert(Task task);  
    void update(Task task);  
    void delete(Task task);  
    void notifyAllObservers();  
}
```

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

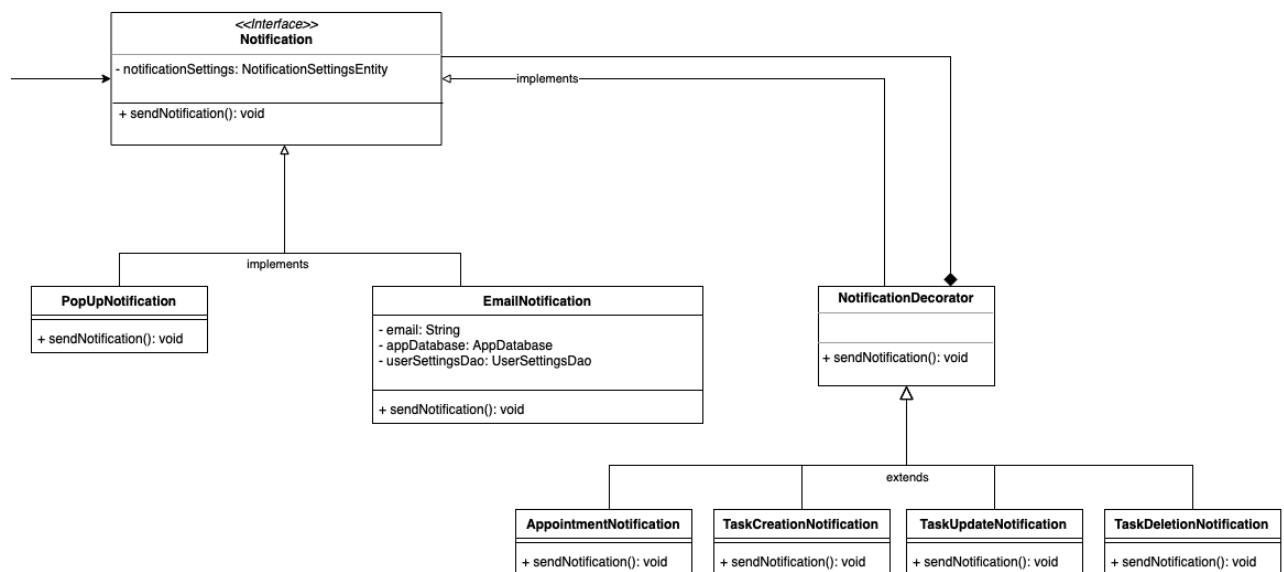
```
public class NotificationObserver extends Observer{  
    private NotificationService notificationService;  
  
    @Override  
    public void update(ActionType actionType, Task task) { super.update(actionType, task); }
```

1.2.6 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.



```

public interface Notification {
    NotificationSettings notificationSettings = null;

    void sendNotification();
}

```

```

public class EmailNotification implements Notification{

    @Override
    public void sendNotification(){}
}

```

```

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

```

```

public class NotificationDecorator implements Notification{
    protected Notification notification;

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

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

```



```
public class AppointmentNotification extends NotificationDecorator{

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

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

}
```

```
public class TaskCreationNotification extends NotificationDecorator{

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

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

}
```

```
public class TaskDeletionNotification extends NotificationDecorator{

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

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

}
```

```
public class TaskUpdateNotification extends NotificationDecorator{

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

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

}
```

1.2.7 Facade Pattern

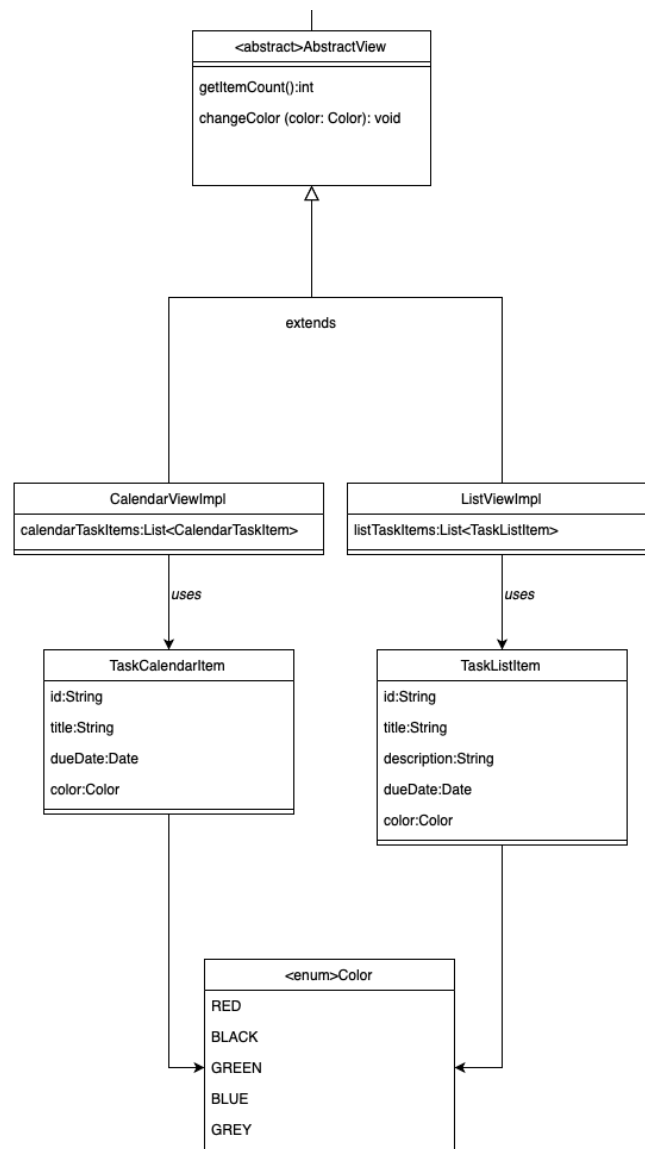
Facade pattern approach:

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.



```

import android.graphics.Color;

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

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

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

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

```
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; }
}
```

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

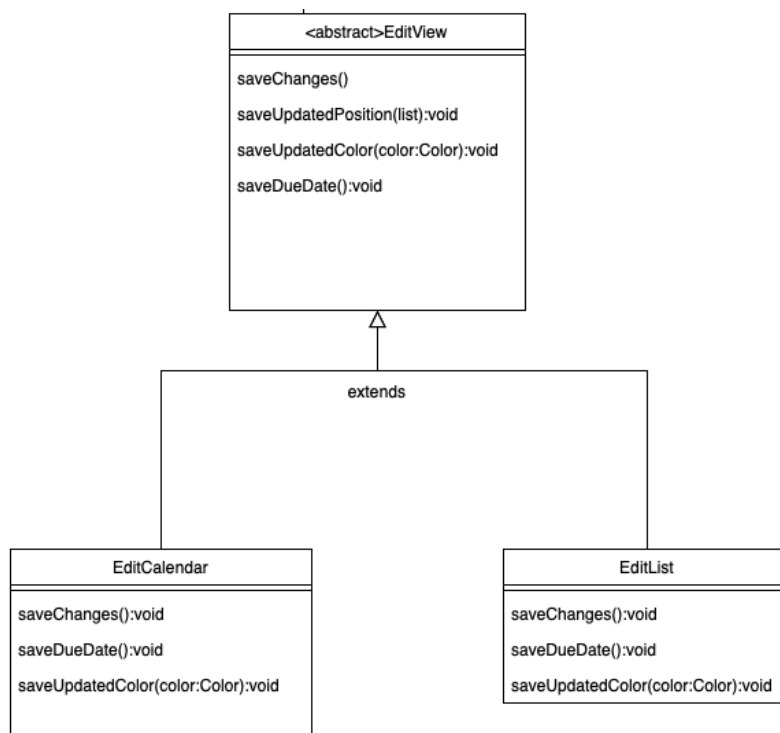
```
public class CalendarTaskItem {  
    private String id;  
    private String title;  
    private Date dueDate;  
    private EColor eColor;  
}
```

```
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; }  
}
```

1.2.8 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() {

    }
}
```

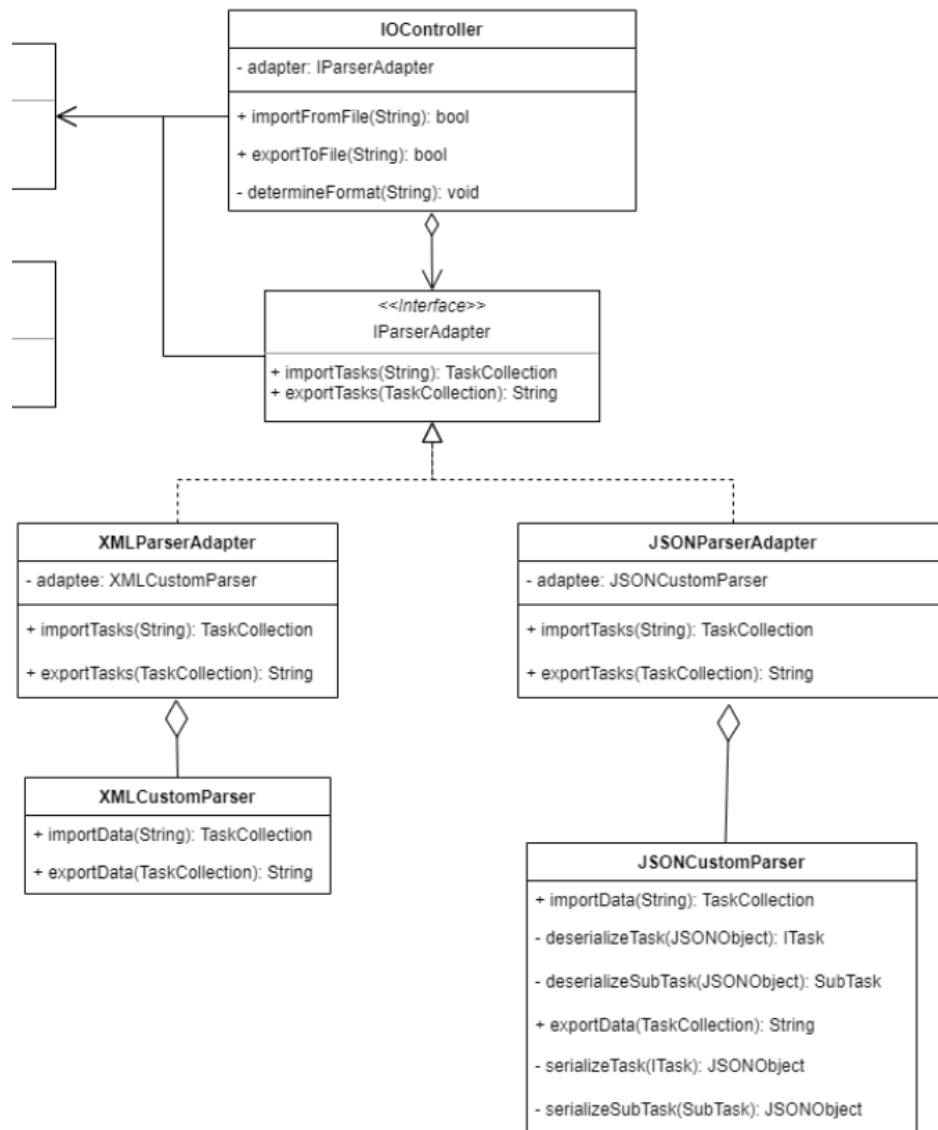
```
public class EditCalendar extends EditView {  
    @Override  
    public void saveChanges() { super.saveChanges(); }  
  
    @Override  
    public void saveUpdatedPosition() { super.saveUpdatedPosition(); }  
  
    @Override  
    public void saveUpdatedColor(EColor color) { super.saveUpdatedColor(color); }  
  
    @Override  
    public void saveDueDate() { super.saveDueDate(); }  
}
```

```
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(); }  
}
```

1.2.9 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 strings. 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);
    }

}
```

```

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;
    }

}

```

```

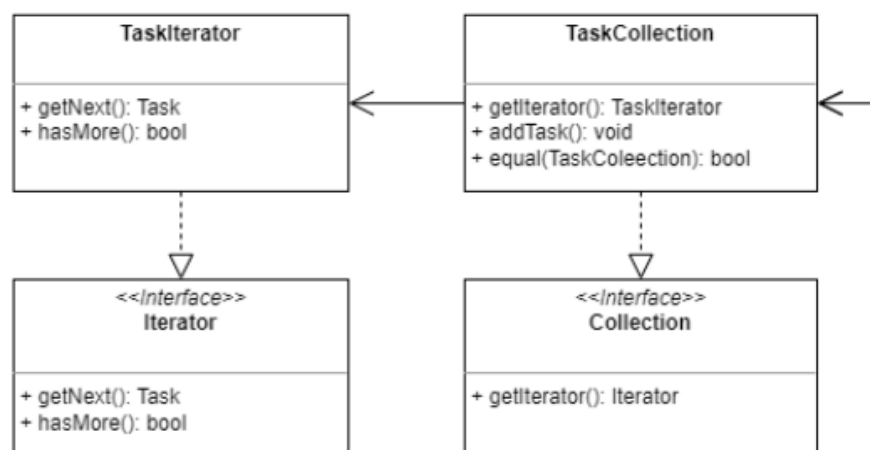
/**
 * Method determines the format of string to be deserialized from or
 * to be serialized to. Creates either JSON or XML parser and
 * assign it to class attribute 'adapter'. As a result,
 * other methods are not aware of type of parser,
 * that they are using.
 * @param string Formatted string, whose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

```

1.2.10 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.

```

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.

```

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 Code Metrics

The following screenshot shows the overview of source code.

The project contains:

- 79 Java classes with 1462 lines of code in total.
- 25 XML files with 773 lines of code in total.

Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines	Lines MIN	Lines MAX	Lines AVG	Lines CODE
bat (BAT files)	1x	2kB	2kB	2kB	2kB	89	89	89	89	68
gitignore (GITIGNORE files)	2x	0kB	0kB	0kB	0kB	18	1	17	9	17
gradle (GRADLE files)	2x	0kB	0kB	0kB	0kB	21	5	16	10	20
java (Java classes)	79x	55kB	0kB	6kB	0kB	2140	4	145	27	1462
md (MD files)	1x	0kB	0kB	0kB	0kB	1	1	1	1	1
pro (PRO files)	1x	0kB	0kB	0kB	0kB	21	21	21	21	0
properties (Java properties files)	3x	1kB	0kB	1kB	0kB	35	6	21	11	9
webp (WEBP files)	10x	33kB	0kB	7kB	3kB	250	6	58	25	249
xml (XML configuration file)	25x	33kB	0kB	7kB	1kB	886	3	198	35	773
Total:	124x	128kB	5kB	26kB	10kB	3461	136	566	228	

Fig: Code metrics Overview

The following screenshot represents the metrics for java classes, where

- 2140 total lines
- 1462 source code lines
- 149 comment lines
- 529 blank lines

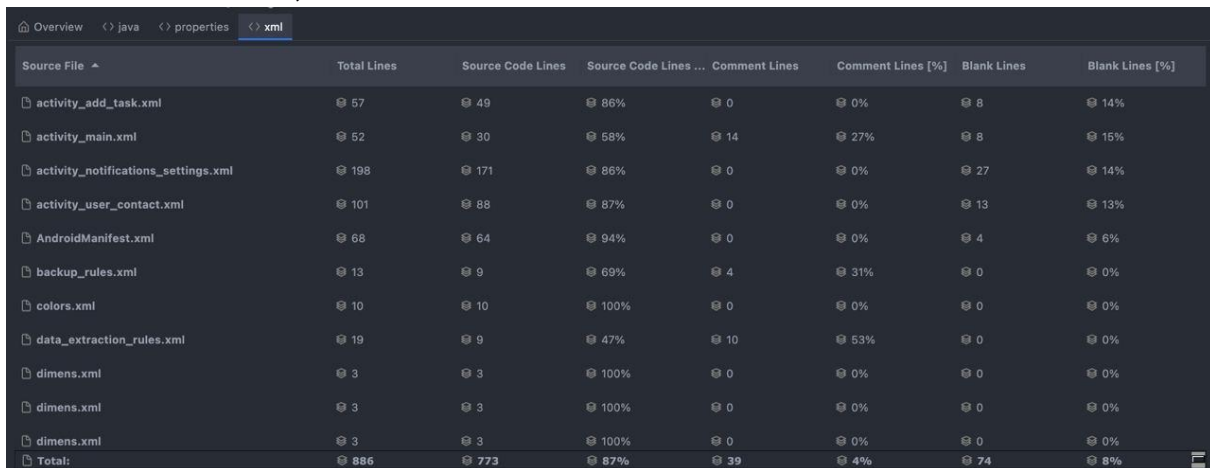
Source File	Total Lines	Source Code Lines	Source Code Lines ...	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
AbstractView.java	8	6	75%	0	0%	2	25%
ActionType.java	5	4	80%	0	0%	1	20%
AddTaskActivity.java	103	72	70%	7	7%	24	23%
AppDatabase.java	59	33	56%	15	25%	11	19%
Appointment.java	50	36	72%	0	0%	14	28%
AppointmentNotification.java	13	10	77%	0	0%	3	23%
AttachmentA.java	11	8	73%	1	9%	2	18%
AttachmentB.java	11	8	73%	1	9%	2	18%
AttachmentFormat.java	5	4	80%	0	0%	1	20%
CalendarTaskItem.java	10	8	80%	0	0%	2	20%
CalendarViewImpl.java	19	14	74%	0	0%	5	26%
Total:	2140	1462	68%	149	7%	529	25%

Fig: Java Classes

The metrics for the XML files are shown below:

- 886 total lines;
- 773 source code lines;

- 39 comment lines;
- 74 blank lines;



Source File	Total Lines	Source Code Lines	Source Code Lines ...	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
activity_add_task.xml	57	49	86%	0	0%	8	14%
activity_main.xml	52	30	58%	14	27%	8	15%
activity_notifications_settings.xml	198	171	86%	0	0%	27	14%
activity_user_contact.xml	101	88	87%	0	0%	13	13%
AndroidManifest.xml	68	64	94%	0	0%	4	6%
backup_rules.xml	13	9	69%	4	31%	0	0%
colors.xml	10	10	100%	0	0%	0	0%
data_extraction_rules.xml	19	9	47%	10	53%	0	0%
dimens.xml	3	3	100%	0	0%	0	0%
dimens.xml	3	3	100%	0	0%	0	0%
dimens.xml	3	3	100%	0	0%	0	0%
Total:	886	773	87%	39	4%	74	8%

Fig: XML files metrics

Regard to the current state of a project there are:

- 440 warnings, 335 of warnings are in Java Classes;
- 4 errors;

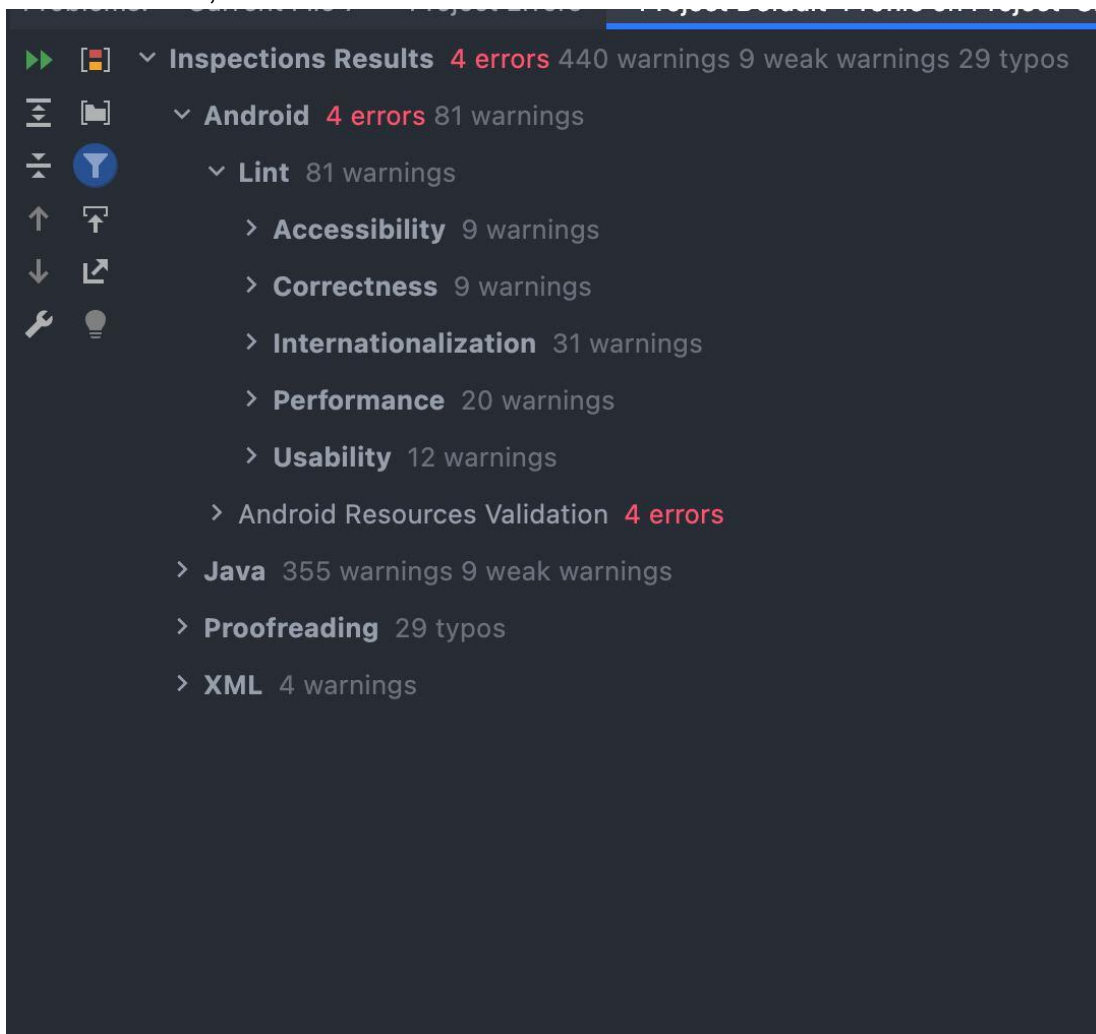


Fig: Code bugs

The project has three packages:

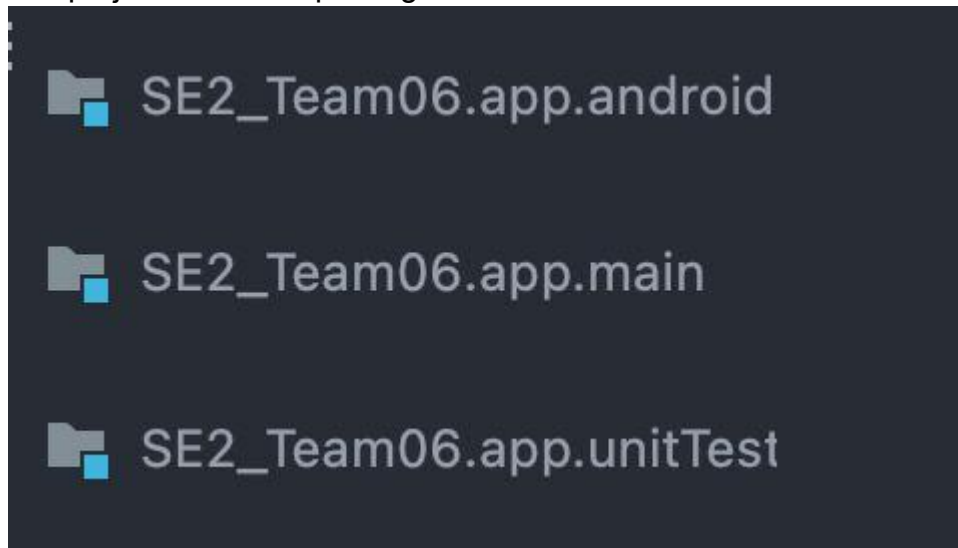
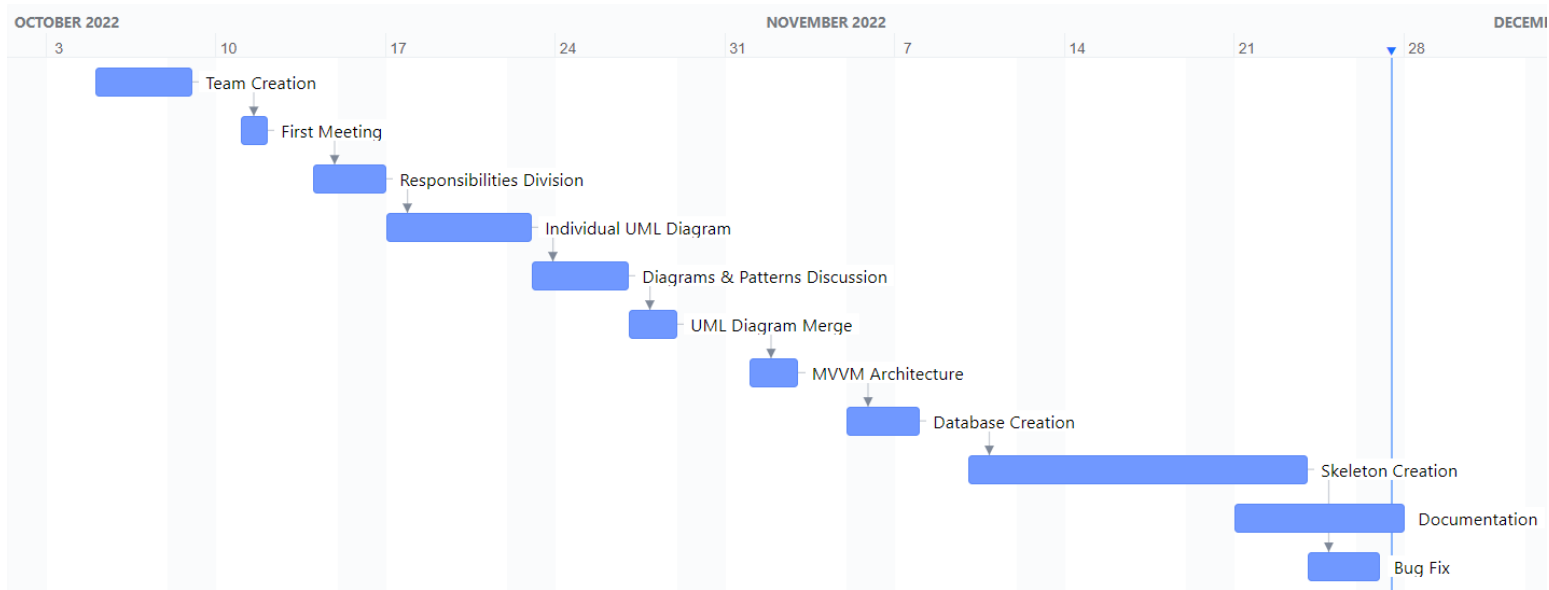


Fig: Project Packages

3 Team Contribution

3.1 Project Tasks and Schedule

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



3.2 Distribution of Work and Efforts

Contribution of member 1:

Member 1 is responsible for the basic functionality for the first deadline. Since Task creation should be implemented by them. Member 1 set up the Project, created the database (using Room, and with help of Member 3), created necessary Views, Models and viewModels to have an UI for creating an Appointment Task and saving it in the database.

They contributed to this Documentation with their planned Design Patterns (Factory and Proxy Pattern).

Most time was spent, of this member, was coding and debugging the Android Studio Project.

setting up project and uploading to git: 1h

Create UMLs: 10h

Designing Patterns: 10h

Writing code: 30h

Debugging: 20h

Writing documentation: 3h

Contribution of member 2:

Member 2 implementation, Documentation

Design patterns: Composite and Strategy.

Contribution of member 3:

Member 3 implementation

Documentation of application components for database, class diagram overview, technology stack, individual parts for design patterns description - 5h

Design patterns: Decorator, Observer, Factory (for notification functionality) - 15h

Skeleton of implementation of notifications related logic with using design patterns - 10h

Set up Room database - 5h

Created basic UI for notification and account settings - 3h

Design UML diagram - 10h

Contribution of member 4:

Documentation of individual parts - 4 h

Code metrics - 1 h;

Implement design patterns: Template Method, Facade Pattern - 10 h;

Skeleton Implementation: 10h;

Uml diagram - 10h ;

Contribution of member 5:

Member 5 Implementation

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*