Software Engineering 2
DESIGN REPORT

Team number:	0206

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

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

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

Team member 4	
Name:	Mohammad Mahdi
Student ID:	01428941
E-mail address:	a01428941@unet.univie.ac.at

Team member 5	
Name:	Sahel Naderi
Student ID:	11823369
E-mail address:	a11823360@unet.univie.ac.at

1 Design Draft

1.1 Design Approach and Overview

(Note: all class diagrams were drawn with the tool "draw.io", since the website supports zoom in and out for details the diagram in this document might be blurred and unreadable, therefore all the diagrams in less uncompressed form will be provided in the MODEL folder too)

We started the design progress by studying and researching patterns together. It is important and beneficial to exchange technical ideas and get everyone on the same page.

Then, we arranged group meetings to brainstorm different ideas together on a high level abstraction (see Figure 1.1) of where to plant the patterns and the major classes we needed. Although this approach might take longer than splitting up the responsibility, more mistakes could be identified during this stage. (since there are 5 of us putting our thoughts together on one section)

As we progressed, the class diagram was refined into low level structure with more details on operations and patterns. There were some reiterations already, however we will only show the latest UML here (see Figure 1.2), and the design decisions will be discussed in the next section.

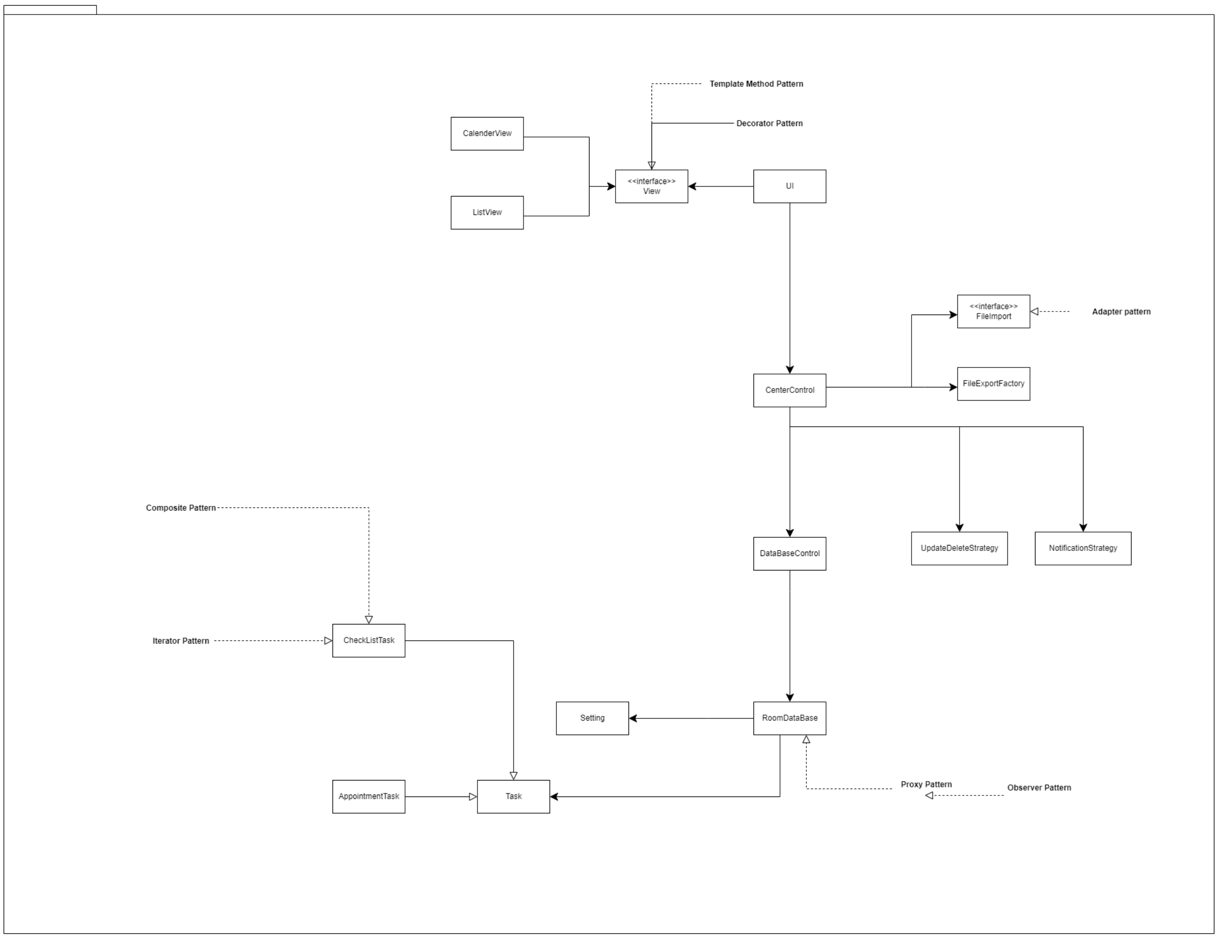


Figure 1.1: high level structure design (note: the high level structure were being modified couple of times, this is the newest version)

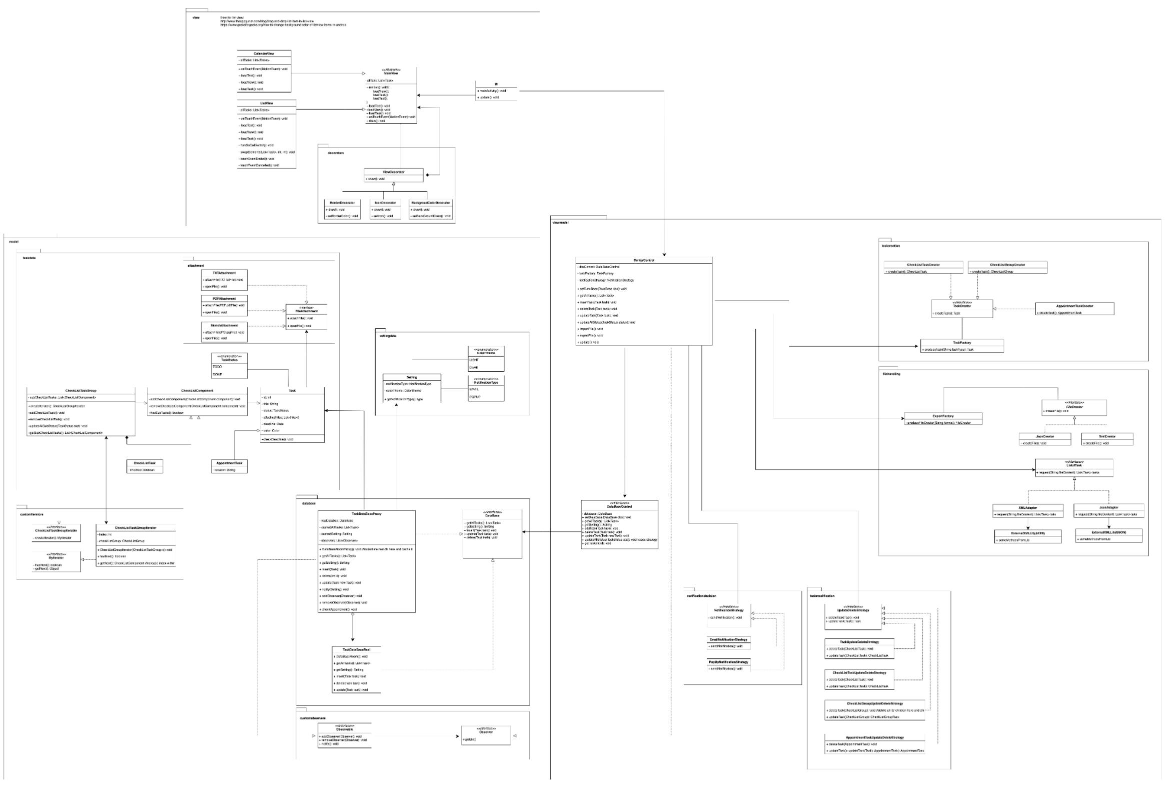


Figure 1.2: low level view of entire software design

Design Decisions:

Task creation: We decided on Factory pattern over Strategy pattern.

Reason: Looking into the concepts and definition of the Factory pattern, it is more suitable in the situation of creating objects rather than using different algorithms at runtimes, although both patterns would work to solve the problem.

File exportation: We decided on Factory pattern over Strategy and Adapter pattern.

Reason: In our opinion, to export a File(XML, Json) is to create Files (objects), which is corresponding to Factory and the pattern implementation itself is very simple and easy to understand.

However Strategy and Adapter pattern could work here too. We can use a different strategy based on which file format the user wants to export, or for the Adapter pattern we can use some external library thus use an Adapter to connect the outside function to fit into our structure without the modification of the original function.

File importation: We decided on Adapter pattern over Strategy and Factory pattern.

For importation of the tasks the Adapter acts as a kind of intermediate between the external libraries for importation and the client (CenterControl), who requests the import.

We could have also used the Factory or Strategy here for the same reasons, described in the “File exportation” section above.

Sub task and subtasks handling: We decided on idea 2 (see Figure 2.2).

Reason: Idea 2 seems simpler and more logical to us at first glance (it also came to our mind first), after deeper thoughts we found out that if we choose the first idea, the deletion and the handling of subtasks will be more complex, also the performance is worse than idea 2.

The Difference between the two ideas is that, idea 1 puts a Composite pattern on the Task class (see Figure 2.1), which means a group of tasks can consist of both appointments or checklists (note: appointment and checklist are two types of tasks). Where as idea 2, the Composite pattern sits on the checklist, thus the group could now only consist of checklists, thus handling of subtasks will only need to be considered on the checklists and not appointments.

(Note: Figure 2.1 and 2.2 are very high level and at the time we still use strategy pattern on CURD, later on we separate the creation of tasks into Factory pattern.)

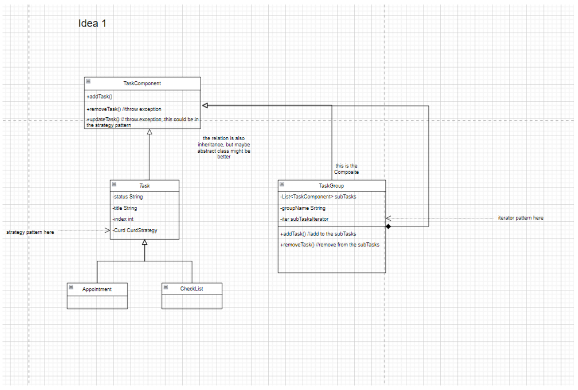

Idea 1
TaskComponent
+addTask()
+removeTask() //throw exception
+updateTask() // throw exception, this could be in the strategy pattern
the relation is also inheritance, but maybe abstract class might be better
this is the Composite
Task
-status String
-title String
-index int
-Curd CurdStrategy
strategy pattern here
TaskGroup
-List<TaskComponent> subTasks
-groupName Srtring
-iter subTasksIterator
+addTask() //add to the subTasks
+removeTask() //remove from the subTasks
iterator pattern here
Appointment
CheckList

Figure 2.1: Composite pattern design idea 1

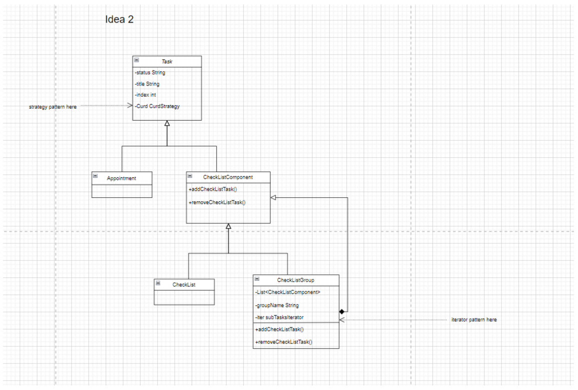


Figure 2.2: Composite pattern design idea 2

Observer location when something happens in the database: We decided to put the Observer on the proxy of the database, rather than the real dbs. (detailed Proxy pattern explained in later section).

Reason: If the observer sits in the real dbs, and when notifying the client after the task inserted/updated/deleted will have delays. It will affect the user experience, but if the observer sits in the proxy where the data is cached "locally", it will perform faster than the dbs query.

Deletion/Updating of parent and/or subtasks: We decided in this case on the Strategy pattern over the Iterator pattern.

Reason: We chose the Strategy pattern because we have different types of tasks that can be used. The different types of tasks (e.g. checklist and appointment) result in similar classes that differ only in the way they execute a certain behavior, in this case deleting tasks.

Thus, we would like to extract these different behaviors into a separate class hierarchy based on the Strategy pattern and combine the original classes into one, thus avoiding duplicate code.

Why we chose Strategy and not Iterator: We have found in the process of our discussions that with an iterator it could be difficult to tell if it is a child or if it is a parent. If we had used Iterator for this case, we think it would be much too complicated to implement. It would be possible, of course, but it would go beyond the scope. In the context of this task it is sufficient with Strategy-Pattern to implement only one method per task type. Another reason would be the performance. Instead of iterating and deleting with Iterator, the algorithm defined for the task type is used and the selected tasks can be deleted.

Notification decision: We decided to combine Observer and Strategy pattern

Reason: Since new notification needs to be created whenever some change happens, for example new task is created or updates it makes sense to implement Observer which will listen to any changes and notify the subscribers. In our case, when there is a change in the database, central control will be notified and invoke the action to respond to the notification. The action in our project will be executing one of the notification strategies based on what is previously selected as a preferred notification type in the Settings.

Our initial idea for creating notifications was to use the Factory pattern but we realized that we actually don't need to create new objects but only implement different algorithms based on the which notification type user selected. That's why we switched to the Strategy pattern.

1.1.1 Class Diagrams

🟧Member 1, 🟦Member 2, 🟥Member 3, 🟪Member 4, 🟪Member 5

(*Note: different color highlighting represents member responsibility*)

Package model (see Figure 3.1):

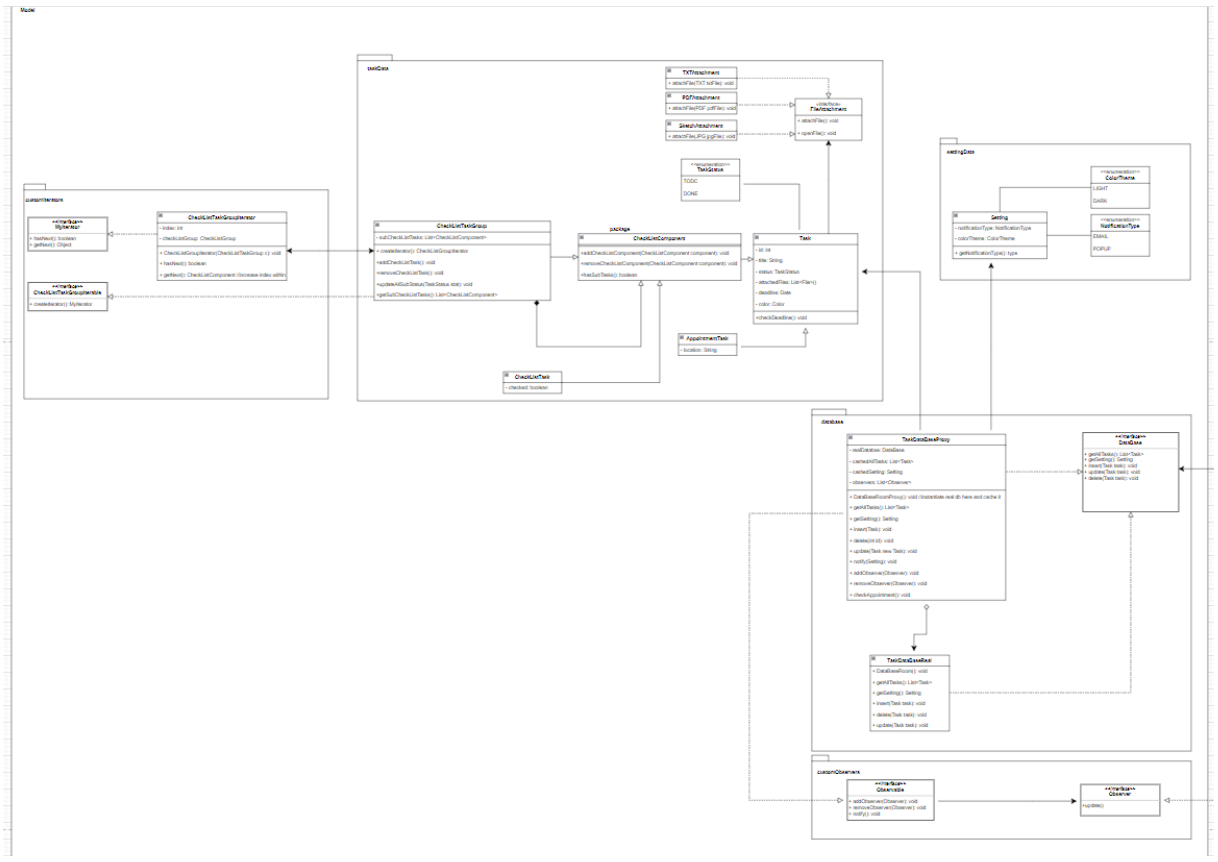


Figure 3.1: detailed view of the model package

- subpackage customiterators 🟧🟦

Contains all the necessary classes for iterating through a group of checklists (-subCheckListTasks: List<CheckListComponent>, detailed pattern explanation will be in section 1.2). It is consider being used to handle the subtasks, mainly on updating the status of all of them requested in Member 1's reponsibility.

- subpackage taskdata 🟧🟦

Contains all the necessary classes for task. There are 2 levels of inheritance here, Task as the parent class, AppointmentTask and CheckListComponent as the children. Then CheckListComponent is a parent class too, it's children are CheckListTask and CheckListTaskGroup (detailed Composite pattern explanation will be in section 1.2). Task class contains all the common properties, including (but not limited to) TaskStatus, stored as an Enum. a List of attached files (-attachedFiles: List<File>), where client could use the FileAttachment to add different files (Member 2's repsonsibility). Also the hand drawn sketch will be stores as PNG here.

- subpackage settingdata 🟥🟪

Contains all the necessary classes for the setting data. The Setting class is to stores the notification type user chose (Memeber 3's responsibilty), also the theme setting of the entire app (dark/light, Member 4's responsibility). Both type will be stored as Enum.

- subpackage database 🟧

Contains all the necessary classes for dbs. This package here will handles the datastorage. It has 2 dbs, one act as a proxy (TaskDataBaseProxy), the other is the real one (TaskDataBaseReal). The proxy will have the cached tasks (- cashedAllTasks: List<Task>), setting(- cashedSetting: Setting) and the observers (- observers: List<Observer>), where the real one does not. The observers here is to notify the client something has changed in the cached data (e.g. insert a new task)

Both implements interface DataBase that will be used by client, so that client does not need to know which database its using or how its using. (detailed Proxy pattern will be explained in section 1.2)

- subpackage customobservers 🟥

Contains all the necessary classes for Observer pattern. It is used to first, notifying the client the changing of cached data, second notifying an up coming appointment. Everytime the app the opened, it caches the data into the proxy, soon as the caching process finished, + checkAppointment(): void will be called to use the observer here to notify the client.

Update: after implementation of Room Database, the observer will not notify the changing of cached data anymore, since LiveData library has it's own observer implemented, thus client only needs to override the methods.

Package viewmodel (see Figure 3.2):

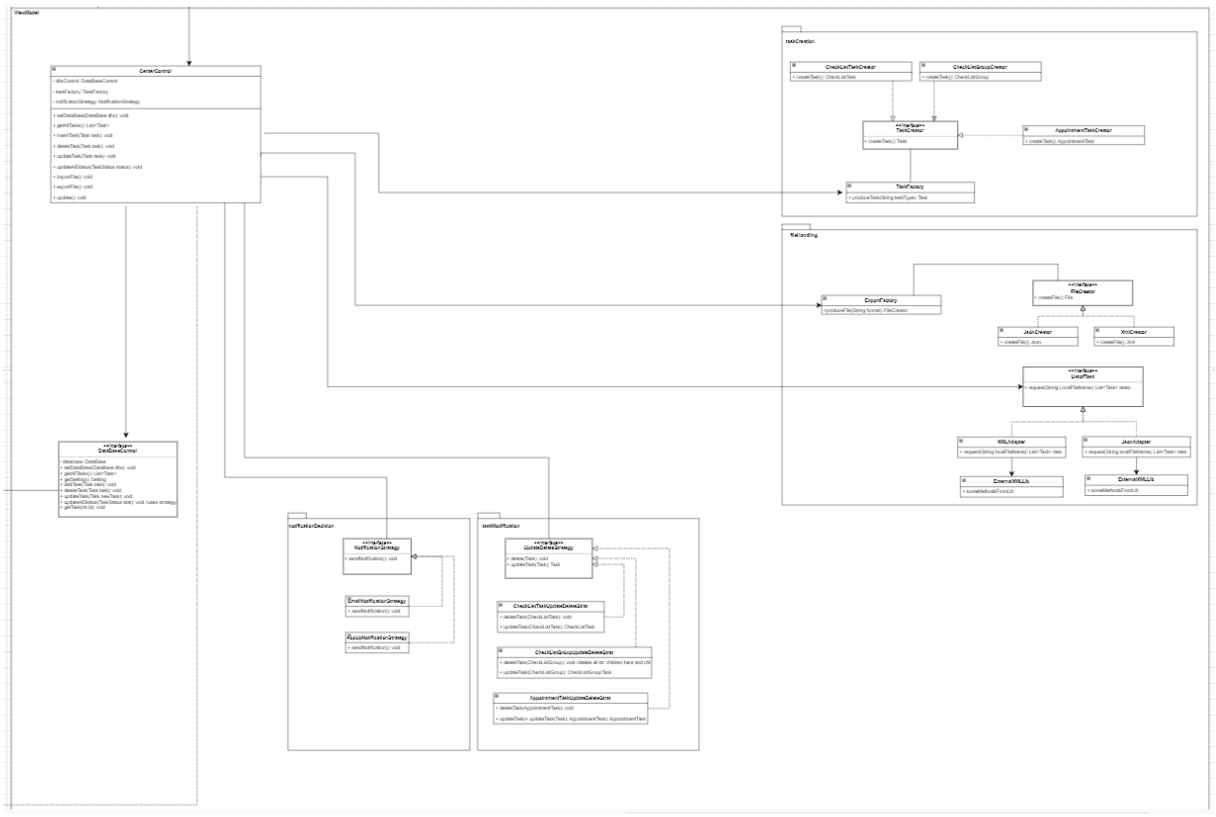

ViewModel
CenterControl
- dbsControl: DataBaseControl
- taskFactory: TaskFactory
- notificationStrategy: NotificationStrategy
+ setDataBase(DataBase dbs): void
+ getAllTasks(): List<Task>
+ insertTask(Task task): void
+ deleteTask(Task task): void
+ updateTask(Task task): void
+ updateAllStatus(TaskStatus status): void
+ importFile(): void
+ exportFile(): void
+ update(): void
<<interface>> DataBaseControl
- database: DataBase
+ setDataBase(DataBase dbs): void
+ getAllTasks(): List<Task>
+ getSetting(): Setting
+ addTask(Task task): void
+ deleteTask(Task task): void
+ updateTask(Task newTask): void
+ updateAllStatus(TaskStatus stat): void //uses strategy
+ getTask(int id): void
taskCreation
CheckListTaskCreator
+ createTask(): CheckListTask
CheckListGroupCreator
+ createTask(): CheckListGroup
<<interface>> TaskCreator
+ createTask(): Task
AppointmentTaskCreator
+ createTask(): AppointmentTask
TaskFactory
+ produceTask(String taskType): Task
fileHandling
ExportFactory
+produceFile(String format): FileCreator
<<interface>> FileCreator
+ createFile(): File
JsonCreator
+ createFile(): Json
XmlCreator
+ createFile(): Xml
<<interface>> ListofTask
+ request(String LocalFileName): List<Task> tasks
XMLAdapter
+ request(String localFileName): List<Task> tasks
JsonAdapter
+ request(String localFileName): List<Task> tasks
ExternalXMLLib
+ someMethodsFromLib
ExternalXMLLib
+ someMethodsFromLib
notificationDecision
<<interface>> NotificationStrategy
+ sendNotification(): void
EmailNotificationStrategy
+ sendNotification(): void
PopUpNotificationStrategy
+ sendNotification(): void
taskModification
<<interface>> UpdateDeleteStrategy
+ delete(Task): void
+ updateTask(Task): Task
CheckListTaskUpdateDeleteStrat
+ deleteTask(CheckListTask): void
+ updateTask(CheckListTask): CheckListTask
CheckListGroupUpdateDeleteStrat
+ deleteTask(CheckListGroup): void //delete all its' children here and chi
+ updateTask(CheckListGroup): CheckListGroupTask
AppointmentTaskUpdateDeleteStrat
+ deleteTask(AppointmentTask): void
+ updateTask(+ updateTask(Task): AppointmentTask): AppointmentTask

Figure 3.2: detailed view of the viewmodel package

- interface DataBaseControl: 🟧

Has a instance of DataBase interface (`- database: DataBase`) in database package to control the data. This instance needs to be set in the begining, in order to decides which dbs should be used (real/proxy). It should also handle the concatenation of data (if different task type stored in different tables). In the future if there are data coming from outside (api, web app. , etc.), they should be handled here too.

- class CenterControl: 🟧🟦🟥🟪🟪

Should act as the “connector” between UI and the backend. It act as a Facade pattern and has the control to everything, including modification of tasks, sending notifications, creation of task and export/import from external file.

- subpackage notificationdecision: 🟥

Contains different strategies to send out whether emails or a warning message.

- subpackage taskmodification: 🟧🟦

Contains different strategies to update or delete different task types.

- subpackage filehandling: 🟪

Contains all necessary classes for exporting and importing Json/Xml files.

- subpackage taskcreation: 🟧

Contains all necessary classes for handling the creation of different tasks.

Package view (see Figure 3.3):

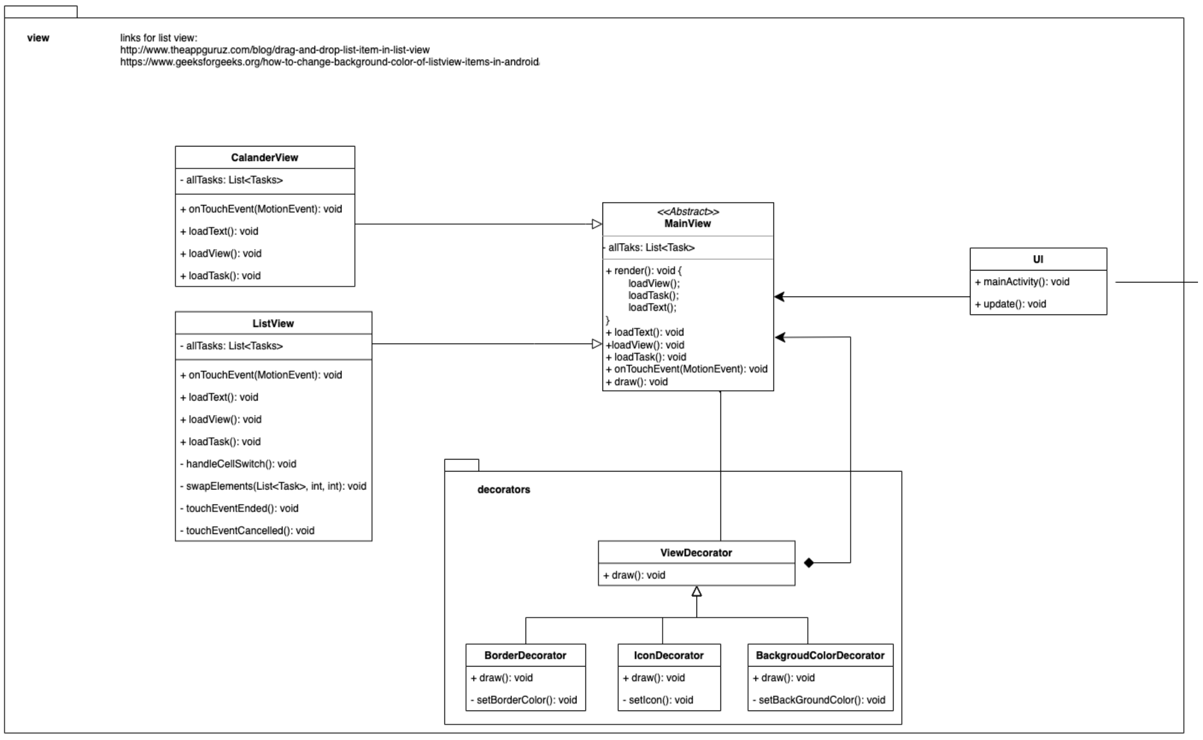


Figure 3.3: detailed view of the view package

- abstract class View: 🟪

Is an abstract class that has one major method: `+ render(): void` that contains all the necessary substeps to render a view (calender/list). It act as a template (detailed Template method pattern will be explained in section 1.2), so that the classes that extend it only needs to change the detail of the substeps.

- subpackage viewtypes: 🟪

Contains 2 different views, a calender and a list view. The reorder of task via drag-and-drop should be handled here as well.

- subpackage decorations: 🟪

Contains different concrete decorators and decorator to decorate the view (adding a border, Scrollbar etc.).

- highlevel class UI: 🟧🟦🟥🟪🟪

Represents all the UI related java files as well as XMLs (e.g. MainActivity.java, activity_main.xml).

1.1.2 Technology Stack

Jetpack compose: abandoned because it uses Kotlin, as none of us have experience with Kotlin.

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

Android Room: version 2.4.3

Android room youtube tutorial: https://www.youtube.com/watch?v=ARpn-1FPNE4&list=PLrnPJCHvNZuDihTpkRs6SpZhqgBqPU118&index=3 (a playlist that we used as guidance to help build the room database for the first phase)

Alternative potential Android room youtube with jetpack compose:

https://www.youtube.com/watch?v=A8AUtcP0rRs&list=PLQkwcJG4YTCS3AD2C-yWtJUGTYMh5h3Zz&index=1 (handling multiple tables, might be useful since later in implementation there are different type of tasks, and they could be stored in different tables. However this tutorial is using Kotlin, only the structure and logic might be usefull)

https://www.youtube.com/watch?v=lwAvI3WDXBY&list=PLSrm9z4zp4mEPOfZNV9O-crOhoMa0G2-o (room tutorial from scratch but uses Kotlin, might be helpful for providing logic and structural thoughts)

Most important libraries:

RecycleView: version 1.2.1

CardView: version 1.0.0

LiveData: version 2.3.1

CalendarView:

https://www.youtube.com/watch?v=Ba0Q-cK1fJo&list=PL45O_vTnNGfV0MW45PmsmsUOHq1MFtC10&index=2 (possible calendar implementation tutorial)

(note: might be useful, however the calenderView provides very limited modification, a custom calendar might still need to be implemented)

Potential Hand drawing libraries:

external library for sketching: https://github.com/gauravyadav673/HandDraw

implementing drawing youtube tutorial: https://www.youtube.com/watch?v=Pa8HfgBIoBg

Potential File Export/Import library:

XMl: JAXB: https://javaee.github.io/jaxb-v2/

Json: Gson Version: 2.10: https://github.com/google/gson

Extra resources:

https://www.youtube.com/playlist?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc (Pattern explanation: almost all the pattern explanations were learn from this playlist, was extremely helpful)

https://refactoring.guru/design-patterns (Also website we used during researching for patterns)

1.2 Design Patterns

1.2.1 Design Pattern 1: Composite Pattern

When we look at the Member 2's responsibility, "how to handle the deletion of a task that contains substasks, the first thing that comes into our mind is the Composite pattern.

The basic concept of composite pattern is to treat one object or a group of objects the same way. For example you have a folder on Desktop, in that folder there are some files. You don't have to know the properties of a folder or a file, you can put the whole folder into the Recycle bin the same way as an individual file. That what Composite pattern should do, it allows you to treat them all the same way.

In our case, the CheckListTask is the individual objects and a group of CheckListTask (CheckListGroup) is the composite, and client should be able to treat them uniformly.

1.2.2 Design Pattern 2: Iterator Pattern

During the reserach about Compostie pattern, we noticed that there is a always a traverse needed inside it, we saw that there is the potential to put the Iterator pattern there. After more thoughts about how to handle the deletion of a subtasks or update the common properties of a group of CheckListTasks, we decided to put the Iterator pattern on top of the CheckListGroups.

Iterator pattern is very simple, it should iterate through a container and return the item you want. With the use of iterator you can further perform other operations, in our case if we want to update all the CheckList properties inside subCheckListTasks or delete some specific CheckLists in there, the Iterator pattern should be very useful for such scenarios.

This pattern it is very frequently used in Java without noticing, since Java has those built-in functions for iterating through containers, whereas in C++, people usually need to make an iterator first in order to traverse through a container.

1.2.3 Design Pattern 3: Proxy Pattern

The proxy pattern is used for example when a website is loading some large images or contents, it will take a very long time, but if we cache the image and load it from the cache, it will significantly reduces the loading time, thus performance, as well as the user experience, and that is what the proxy pattern should do, it should cache the "heavy stuff" into the proxy, and should not take/load from the actual source, in another word "fake" what the actual thing should do.

In our case, we think it is suitable to put it in the database, where the queries might take time, especially as the database grows larger, the queries become even slower. And to have a proxy would solve this issue by cache all the tasks into container first, and all the updates/inserts/deletes will be performed on the container instead of on the actual database.

1.2.4 Design Pattern 4: Facade Pattern

In our opinion Facade pattern is the easiest and the most straight forwarded pattern out of all the others. The Facade is similar to a controller, where it has the control over everything, and hide all the complexity for client.

In our case, there are a few Facade pattens. The major one is the CenterControl class, its the glue that holds everything together. It has the control over dbs with DataBaseControl interface; sending notification by using NotificationStrategy interface; insert/update/delete/create of different task by using the TaskFactory and the UpdateDeleteStrategy; file export and import.

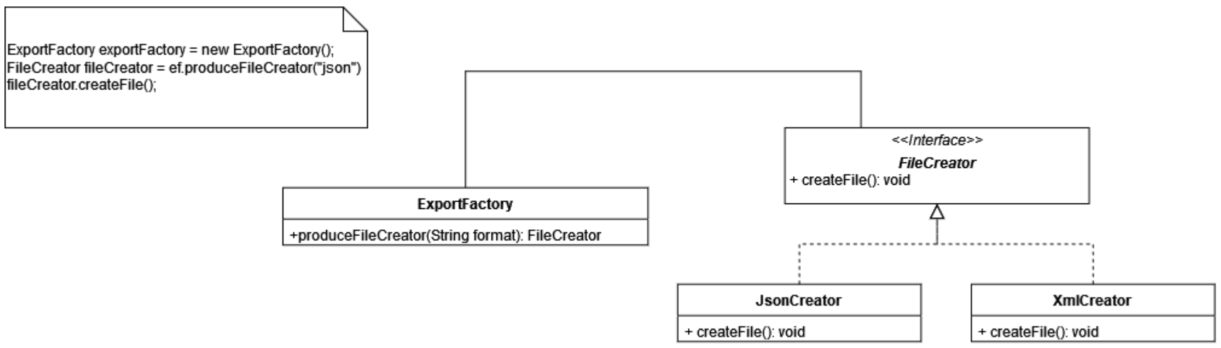
1.2.5 Design Pattern 5: Factory Pattern

Factory pattern as the name suggest, acts like a factory. If someone provides the material to the factory, it will produce different products depending on what material they put it in. In a more technical term, the client put different parameters into the factory, and it will initialise different objects depending on the passed parameter.

In our design, we have 2 Factory patterns:

1. TaskFactory:
   Located in the taskCreation package. Depending on the parameter, it uses different concrete creators (AppointmentTaskCreator, CheckListTaskCreator, CheckListGroupCreator) to create different Task (AppointmentTask, CheckListTask, CheckListGroupTask). It hides the complexity of producing different Tasks, thus client does not need to care how they are created, they just need to "put in the necessary materials".

2. ExportFactory:
   Depending on the passed parameter in the method produceFileCreator of the class "ExportFactory" the client (CenterControl) gets an implementation of the FileCreator interface returned. Hence the client can call the implemented export method, which in our case either creates a Json file, by the JsonCreator class or a XML file, created by the XmlCreator class. As stated before, this way the client(CenterControl) just calls the createFile() from the "FileCreator" interface and does not know about the concrete implementations.



```
ExportFactory exportFactory = new ExportFactory();
FileCreator fileCreator = ef.produceFileCreator("json")
fileCreator.createFile();
```

1.2.6 Design Pattern 6: Strategy Pattern

The strategy pattern comes under the behavior patterns. If we have a class that does something but in a lot of different ways, it makes sense to create classes where we separate all these different algorithms. In our case, we are planning to use the strategy pattern for implementing the notifications. If we had just one Notification class, we would need to implement different algorithms for Email notification and Pop-up notification inside that one class. Then, if we wanted to add more notification types, such as SMS notification and a few others, then we would be adding a new algorithm for each notification and the main class will doubled in size. At some point, it would become really hard to maintain. In addition, any change to one of the algorithms can affect the whole class, increasing the chance of creating an error in already-working code. This is exactly the issue that strategy pattern helps us solve.

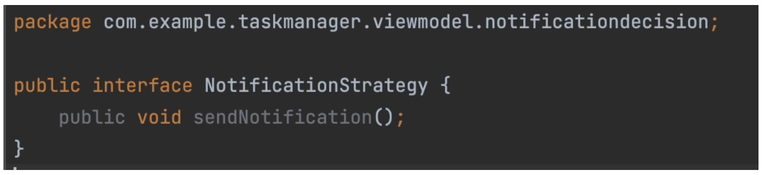
The structure of the strategy pattern is the following:

- Context class - CentralControl class
- Strategy interface - NotificationStrategy class
- Concrete Strategy - EmailNotificationStrategy and PopUpNotificationStrategy class

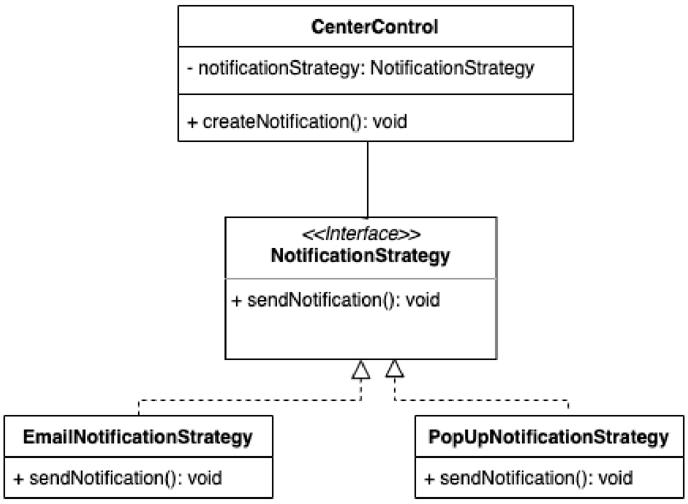
In our project, the CentralControl class maintain the reference to EmailNotificationStrategy and PopUpNotificationStrategy but it communicates with these objects only via NotificationStrategy interface.

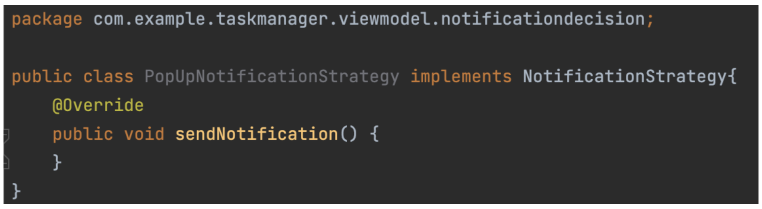
The NotificationStrategy class that represents the strategy interface is common to all concrete strategies. It declares a method the context uses to execute a strategy - sendNotification().

Finally, EmailNotificationStrategy implements an algorithm for email type of the notification, and PopUpNotificationStrategy implements an algorithm for pop up type of the notification. These algorithms are now encapsulated and interchangeable. If we decide to add an algorithm for new type of the notification in the future, for example sms type of the notification, we will just simply create a new class SMSNotificationStrategy which will implement the NotificationStrategy interface.

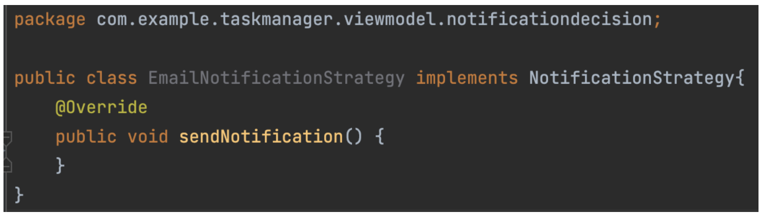


```
package com.example.taskmanager.viewmodel.notificationdecision;

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


CenterControl
- notificationStrategy: NotificationStrategy
+ createNotification(): void
<<Interface>>
NotificationStrategy
+ sendNotification(): void
EmailNotificationStrategy
+ sendNotification(): void
PopUpNotificationStrategy
+ sendNotification(): void



```
package com.example.taskmanager.viewmodel.notificationdecision;

public class PopUpNotificationStrategy implements NotificationStrategy{
    @Override
    public void sendNotification() {
    }
}
```

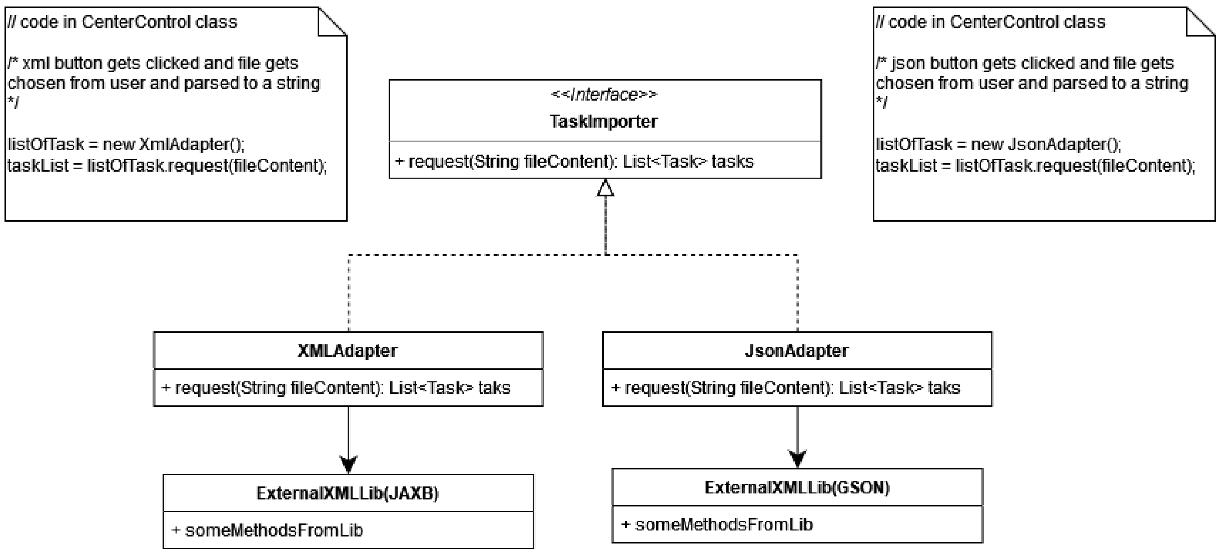


```
package com.example.taskmanager.viewmodel.notificationdecision;

public class EmailNotificationStrategy implements NotificationStrategy{
    @Override
    public void sendNotification() {
    }
}
```

As mentioned in chapter *1.1 Design Approach and Overview* in *Design Decisions* under **Deletion/Updating of parent and/or subtasks**, Strategy Pattern is also very suitable for deletion for tasks and subtasks (Member 2 Responsibility), because there are different types of tasks that can be used. The different types of tasks (e.g. checklist and appointment) result in similar classes that differ only in the way they execute a certain behavior, in this case deleting tasks.

1.2.7 Design Pattern 7: Adapter Pattern

The general idea of Adapter pattern is to make two interfaces, who are incompatible, compatible. To accomplish this, one needs to have an Adapter class, who has access to the incompatible "interface" (which can be a Java class, an external library, an interface etc.), who is the Adaptee. Hence, the Adapter can call methods from the Adaptee, convert them to the format the client expects and provide them to the client.

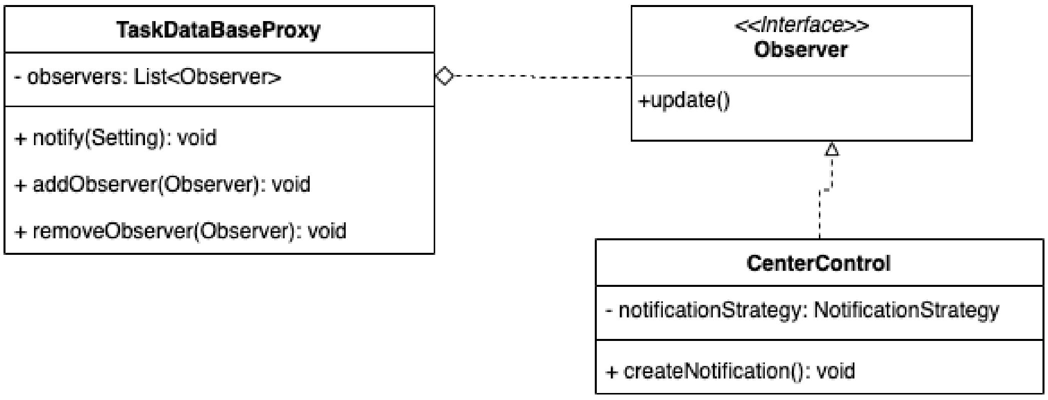

// code in CenterControl class
/* xml button gets clicked and file gets chosen from user and parsed to a string */
listOfTask = new XmlAdapter();
taskList = listOfTask.request(fileContent);
<<Interface>>
TaskImporter
+ request(String fileContent): List<Task> tasks
// code in CenterControl class
/* json button gets clicked and file gets chosen from user and parsed to a string */
listOfTask = new JsonAdapter();
taskList = listOfTask.request(fileContent);
XMLAdapter
+ request(String fileContent): List<Task> taks
JsonAdapter
+ request(String fileContent): List<Task> taks
ExternalXMLLib(JAXB)
+ someMethodsFromLib
ExternalXMLLib(GSON)
+ someMethodsFromLib

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

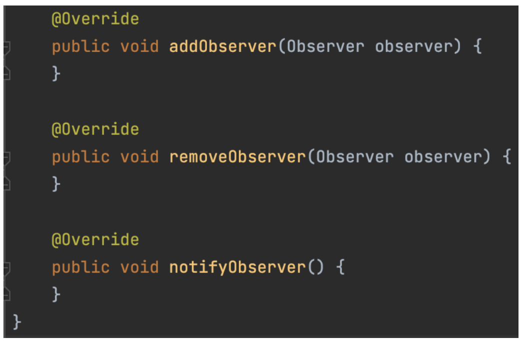
1.2.8 Design Pattern 8: Observer Pattern

Observer is a behavioral design pattern that helps us create a subscription-based communication and enable us to notify multiple subscribers about the events that happen to the object they're observing.

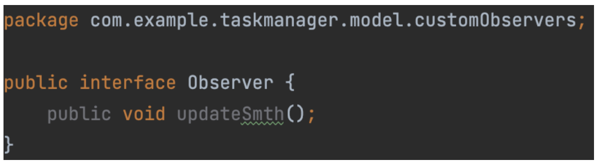
Originally we have the Observer pattern build upon the TaskDataBaseProxy which would in the structure of the pattern represent our Publisher. In this class we have list of the observers which will be notified whenever there's modification happening to the. TaskDatabaseProxy class actually implements the interface called Observable which contains addObserver(Observer), removeObserver(Observer) and notify() methods. Next, the Observer interface is our subscriber which consists of only one method, update(parameter) - parameter should be some event details that are passed from the TaskDataBaseProxy. And finally, our concrete subscriber is CentralControl which will perform some actions in response to notifications issued by the publisher, in our case that would invoke the proper notification to be sent.


TaskDataBaseProxy
- observers: List<Observer>
+ notify(Setting): void
+ addObserver(Observer): void
+ removeObserver(Observer): void
<<Interface>>
Observer
+update()
CenterControl
- notificationStrategy: NotificationStrategy
+ createNotification(): void

However, we found out that the LiveData library has it's own observer implemented already so for create/update/delete tasks we will use LiveData's observer, and our Observer pattern will just handle the upcoming appointment.



```
    @Override
    public void addObserver(Observer observer) {
    }

    @Override
    public void removeObserver(Observer observer) {
    }

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

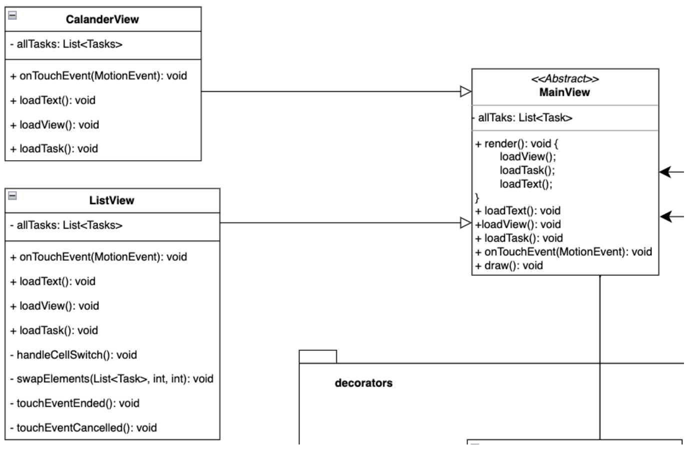


```
package com.example.taskmanager.model.customObservers;

public interface Observer {
    public void updateSmth();
}
```

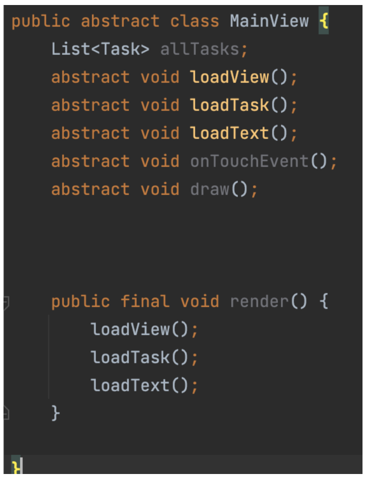
Snippet from TaskDataBaseProxy.java Snippet from Observer.java

1.2.9 Design Pattern 9: Template method Pattern

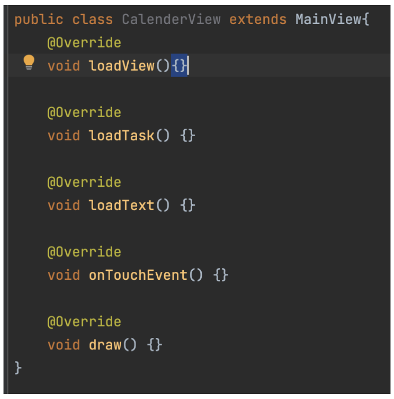
We chose to use this pattern in frontend (view package) so that the ui can interact with both type of views (e.g. calendar and list) with same functions from abstract class called MainView which has a template method called render() that ui can use for accessing custom functionalities of concrete views (CalendarView and ListView)



UI related classes can access concrete views by instantiating the template.



```
public abstract class MainView {
    List<Task> allTasks;
    abstract void loadView();
    abstract void loadTask();
    abstract void loadText();
    abstract void onTouchEvent();
    abstract void draw();



    public final void render() {
        loadView();
        loadTask();
        loadText();
    }

}
```

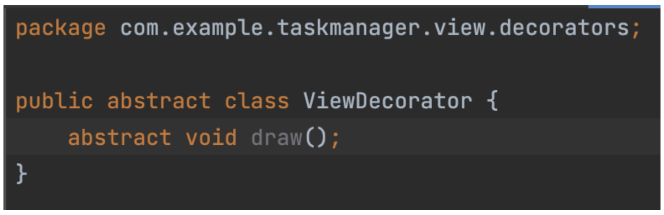


```
public class CalenderView extends MainView{
    @Override
    void loadView(){}

    @Override
    void loadTask() {}

    @Override
    void loadText() {}

    @Override
    void onTouchEvent() {}

    @Override
    void draw() {}
}
```

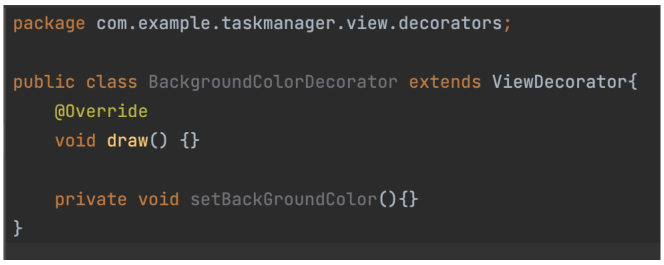
1.2.10 Design Pattern 10: Decorator Pattern

decorator pattern were used in order to add responsibilities to objects not entire classes. in our app we had this idea to add some type of visual cues based on task type, or other task information such as date, etc. For example an upcoming appointment object can have a “attention!” icon, or children of a checklist can inherit their parent background color with a slightly lighter tone, to indicate the parent-child relationship in list view, etc.

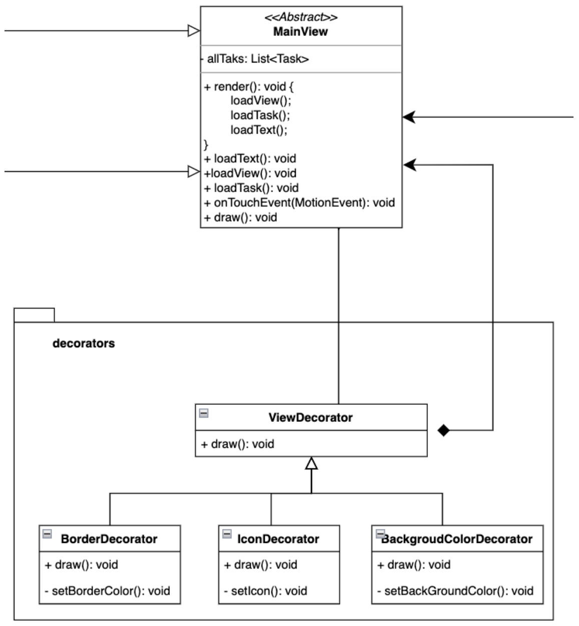
a flexible solution for this is using decorators



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

public abstract class ViewDecorator {
    abstract void draw();
}
```



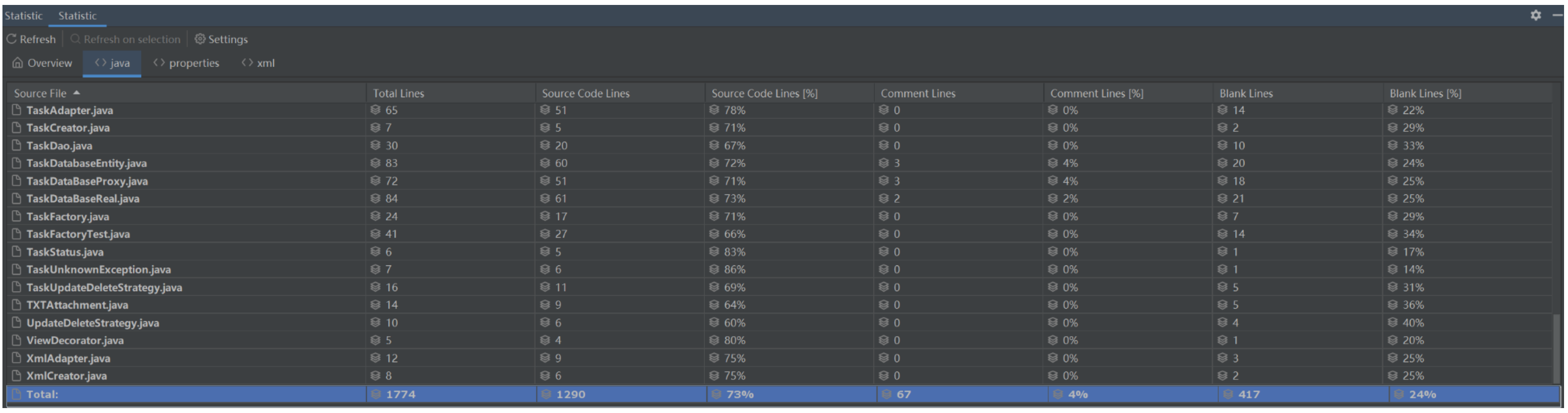
```
package com.example.taskmanager.view.decorators;

public class BackgroundColorDecorator extends ViewDecorator{
    @Override
    void draw() {}

    private void setBackGroundColor(){}
}
```



as you can see the figure above, decorators package which itself resides in view package, we implemented 3 different decorators that each can add a functionality to task objects in view

in project skeleton you can find ViewDecorator with draw function which will be overwritten by 3 specific decorators with each having at least one added functionality

2 Code Metrics



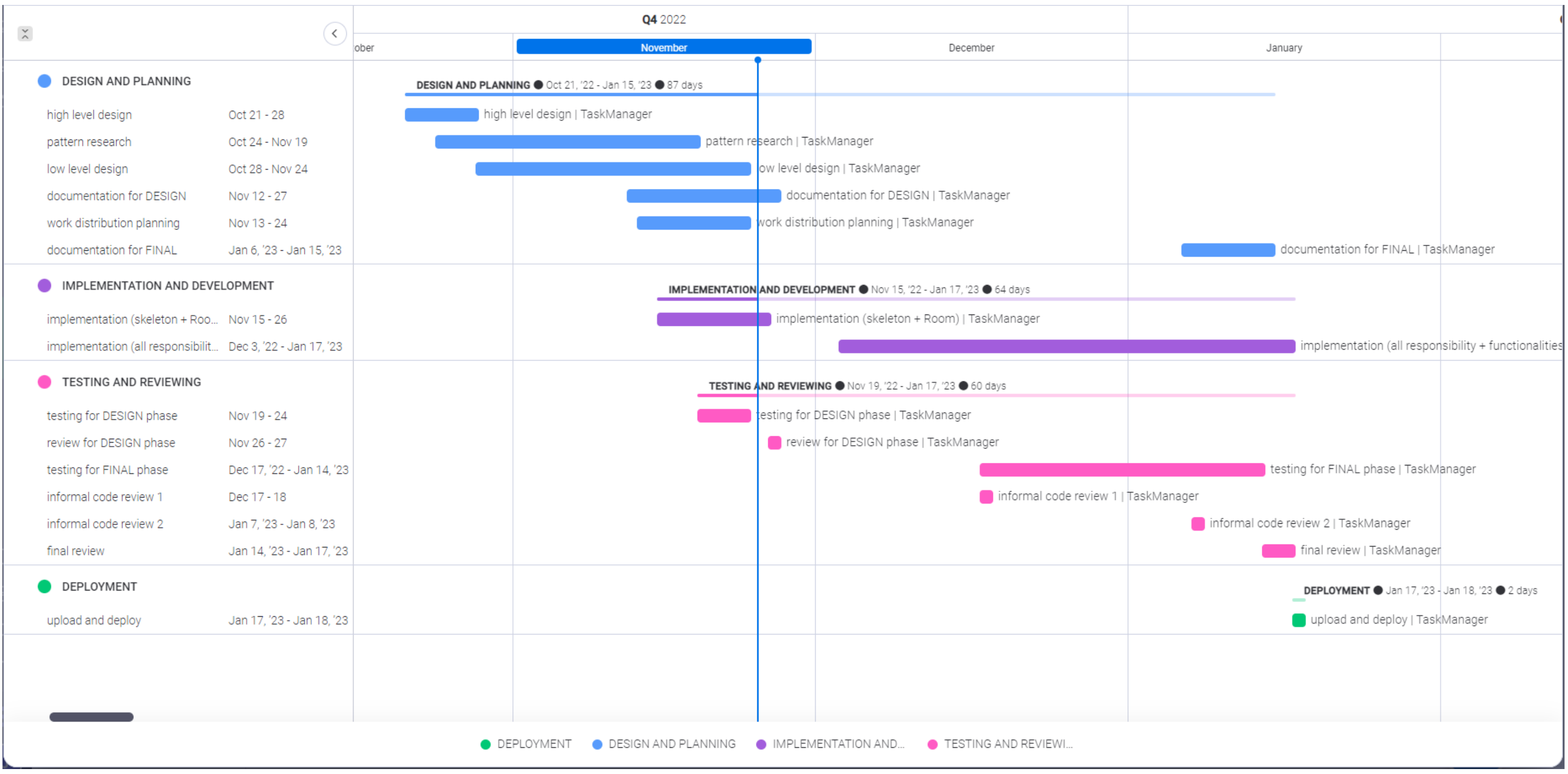
Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
TaskAdapter.java	65	51	78%	0	0%	14	22%
TaskCreator.java	7	5	71%	0	0%	2	29%
TaskDao.java	30	20	67%	0	0%	10	33%
TaskDatabaseEntity.java	83	60	72%	3	4%	20	24%
TaskDataBaseProxy.java	72	51	71%	3	4%	18	25%
TaskDataBaseReal.java	84	61	73%	2	2%	21	25%
TaskFactory.java	24	17	71%	0	0%	7	29%
TaskFactoryTest.java	41	27	66%	0	0%	14	34%
TaskStatus.java	6	5	83%	0	0%	1	17%
TaskUnknownException.java	7	6	86%	0	0%	1	14%
TaskUpdateDeleteStrategy.java	16	11	69%	0	0%	5	31%
TXTAttachment.java	14	9	64%	0	0%	5	36%
UpdateDeleteStrategy.java	10	6	60%	0	0%	4	40%
ViewDecorator.java	5	4	80%	0	0%	1	20%
XmlAdapter.java	12	9	75%	0	0%	3	25%
XmlCreator.java	8	6	75%	0	0%	2	25%
Total:	**1774**	**1290**	**73%**	**67**	**4%**	**417**	**24%**

Current sate:
Total lines of code: 1774,
Commented lines: 67 (4%),
Total classes: 56 (excluding exception classes)
To us the number of classes are acceptable, however more comments needed to be inserted, since more detailed methods need to be done in the future.

Current bugs: if we put an invalid date when creating a new task, the app crashes sometimes when the month and day inputs are invalid. This needs to be fixed in the future.

3 Team Contribution

3.1 Project Tasks and Schedule



DESIGN AND PLANNING	
high level design	Oct 21 - 28
pattern research	Oct 24 - Nov 19
low level design	Oct 28 - Nov 24
documentation for DESIGN	Nov 12 - 27
work distribution planning	Nov 13 - 24
documentation for FINAL	Jan 6, '23 - Jan 15, '23
IMPLEMENTATION AND DEVELOPMENT	
implementation (skeleton + Roo...	Nov 15 - 26
implementation (all responsibilit...	Dec 3, '22 - Jan 17, '23
TESTING AND REVIEWING	
testing for DESIGN phase	Nov 19 - 24
review for DESIGN phase	Nov 26 - 27
testing for FINAL phase	Dec 17, '22 - Jan 14, '23
informal code review 1	Dec 17 - 18
informal code review 2	Jan 7, '23 - Jan 8, '23
final review	Jan 14, '23 - Jan 17, '23
DEPLOYMENT	
upload and deploy	Jan 17, '23 - Jan 18, '23

DESIGN AND PLANNING ● Oct 21, '22 - Jan 15, '23 ● 87 days

IMPLEMENTATION AND DEVELOPMENT ● Nov 15, '22 - Jan 17, '23 ● 64 days

TESTING AND REVIEWING ● Nov 19, '22 - Jan 17, '23 ● 60 days

DEPLOYMENT ● Jan 17, '23 - Jan 18, '23 ● 2 days

Q4 2022
ober
November
December
January
high level design | TaskManager
pattern research | TaskManager
low level design | TaskManager
documentation for DESIGN | TaskManager
work distribution planning | TaskManager
documentation for FINAL | TaskManager
implementation (skeleton + Room) | TaskManager
implementation (all responsibility + functionalities
testing for DESIGN phase | TaskManager
review for DESIGN phase | TaskManager
testing for FINAL phase | TaskManager
informal code review 1 | TaskManager
informal code review 2 | TaskManager
final review | TaskManager
upload and deploy | TaskManager
DEPLOYMENT
DESIGN AND PLANNING
IMPLEMENTATION AND...
TESTING AND REVIEWI...

(Figure 4: Gantt chart for planned schedule. Tool provided by www.monday.com)

(Note: during DESIGN phase we had minimum 2 team meetings per week started from 21st Oct)

3.2 Distribution of Work and Efforts

Contribution of member 1: JIN YU

Research on all patterns: about 2 weeks
UML diagrams: through out 1 month
Room database research + implementation: 2 weeks
Documentation: less than 3 days
Patterns under responsibility: Facade, Factory, Iterator, Composite, Proxy

Contribution of member 2: Merve Sen

Research on all patterns: about 2 weeks
UML diagrams: through out 1 month
Skeleton implementation: 1 week
Documentation: less than 3 days
Patterns under responsibility: Iterator, Composite, Strategy

Contribution of member 3: Edna Mutapcic

Research on all patterns: about 2 weeks
UML diagrams: through out 1 month
Research on android related tech stack, Skeleton implementation: 2 weeks
Documentation: 3 days
Patterns under responsibility: Observer, Strategy

Contribution of member 4: Mohammad Mahdi Fallah

Research on all patterns: about 2 weeks
UML diagrams: through out 1 month
Skeleton implementation, Git Repo preparation, research on android related tech stack: 2 weeks
Documentation: less than 3 days
Patterns under responsibility: Template method, Decorator

Contribution of member 5: Sahel Naderi

Research on all patterns: about 2 weeks
UML diagrams: through out 1 month
Skeleton implementation, research on alternative export/import libraries: 1 week
Documentation: less than 3 days
Patterns under responsibility: Adapter, Factory