StuderaMera

System Design Document (SDD)

Date: 23/10 2020

Authors: Izabell Arvidsson, Julia Jönmark, Hanna Söderström, Ida Dahl

This version overrides all previous versions

Contents

1. Introduction

1.1. Introduction to the application and this document

StuderaMera is an application whose overall purpose is to help students on all levels to be more structured and more effective in their work.

The application will be written in java and CSS and will use the MVVM (Model, View, ViewModel) pattern for the overall structure. Further explanation of why the application will be using MVVM instead of MVC is explained below.

This document will be handling the explanation of how the application will be built to meet a set of technical requirements. The Requirement and Analysis Document (RAD) will be handling what the project is while this document will be handling how the project is built.

1.2. Design goals

The goal is to have a design that is loosely coupled with an external library. The application should also be open for extension; to be able to add new functions and components. The models shouldn't be coupled to the graphical library, JavaFX, the viewmodel should keep these dependencies as small as possible. This is because JavaFX more easily should be able to be swapped out for other graphical libraries.

1.3. Definitions, acronyms and abbreviations

GUI - Graphical user interface
Java - platform independent programming language
JavaFX - A graphical library for the visual part of the application
Java Serialization - a built in mechanism in Java that can store a type of an object and the data stored in that object. This is achieved by writing to a .ser-file (which is a serialization type of file) where the object is stored as a sequence of bytes. The file can then be deserialized by reading from the file.
MVVM - Model-View-ViewModel: A design pattern to follow when structuring applications involving a GUI. The main idea is to split the model, view-model and the view into different packages. The view communicates with the view-model for data-binding and the view-model therefore handles

the interaction between the view and the model. The view-model interacts with the model to get certain information. The view observes the view-model to receive updates that the view should present.
Spinner - A function built in SceneBuilder, which this application uses to build the graphical part, that makes sure that the user can choose between different values to start the timer with.
ScrollPane - A function in SceneBuilder which makes sure that the user can scroll horizontally in the calendar.
Timeline - Makes it possible to update the property values along the progress of time.

2. System architecture

2.1 Overview

The application will be written in java and will be based on the pattern MVVM. All Views-classes are connected to a fxml-file. The View is connected to a ViewModel-class and the ViewModel-class uses information from the Models. The Models does not depend on any View- or ViewModel-class in any way.

The application will be using the MVVM pattern instead of the MVC pattern because the graphical part of the program, the fxml-files, should have a class, in our case the View-classes, that controls the direct communication between the user and the application. The View-classes needs a class which handles the javaFX functionality that the View-class shouldn't handle by itself, our ViewModel-classes. Then the ViewModel-classes should have a class which handles the functionality which has nothing to do with the graphical part of the program. The MVC pattern has a View and a Controller that controls the view but since our application needs a class that controls everything that happens in the view those two classes weren't enough. Our application also needed two types of models, one which could have a connection with the graphical part and one that had nothing to do with it.

2.2 Software decomposition

2.2.1 General

The application consists mainly of five packages. The packages are Views, ViewModels, Models, ObserverInterfaces, Factory and resources.

Views - handles the input from the user and sends the information to the ViewModel-package and its corresponding class for processing. An FXML-file is attached to this View-class which is the graphical part of the View-class. The View-classes contain the major part of the javaFX that is used in the application.

ViewModel - processes information from the View-class and interacts with the Model-package to get information. Sends back the processed information to the view-class through the observer-pattern to avoid circular dependencies.

Model - where the main calculations are made to later be broadcast back to the View. Has nothing to do with javaFX. Contains pure functionality. Aren't depending on any other class since the model is the base of the application.

ObserverInterfaces - information is sended from the View to the View-model to be updated or handled and then sent back to the view to be displayed graphically through interfaces to get looser dependencies and avoid circular dependencies.

Factory - consist of an interface and a class that handles the loading between different fxml-files. This is for reducing duplicated code throughout the whole program. Without implementing the Factory pattern the View-classes would have had the exact same methods for loading in new fxml-files (except the methodname and the new fxml-file that is supposed to be loaded in).

resources - where the images, CSS-file and fxml-files (the graphical part of the application) are.

2.2.2 Decomposition into subsystems

Apart from using IntelliJ IDEA's own subsystems, JavaFX is used to build up the graphic design. The View-classes and the ViewModel-classes depend on the JavaFX library.

2.2.3 A walkthrough of the application

The application will always restart completely when closed, the toDo-lists and calendar will be saved (how that is done will be described later in this document).

When the user runs the application a first side with three choices will appear. Whether the user decides to enter the timer, to-do-list-part or the calendar, a new fxml-file will be loaded in.

1. The user enters the timer part:
   A view with three spinners will appear with presetted values. If the user chooses to change these values they are directly updated in the TimerView-class. The user has several choices from here. Go to the tips page, go to the help page, go back to the first side or start the timer.

   When the user chooses to start the timer the values of the spinners are sent from the TimerView-class to the TimerViewModel-class to be used. Timelines, variables and images are setted and then the timer is started. While the timer is going the different parts of the timerview are updated through the observer pattern.

   If the user ends the program during this process, the values and process will not be saved.

2. The user enters the toDo-list part:
   A view with the different things that has been saved from earlier (if there are such things) are shown along with a button for adding new things to the list. The user has choices to go to from here. Add a new thing to the list, show an already existing thing from the list, go to the calendar, go to timer or go to help.

   If the user chooses to show an already existing task in the list a new view will appear with all the information the user put in the list before, such as the name of the list, checklists, if the checklist is done or not, timeframe and deadline. The values have been saved earlier by Java Serialization and the ToDoListView class accesses this information by observer pattern. The ToDoListView class then fills in all the fields with their matching values. If the user chooses to add a new thing to the list a view with the different fields will be shown and the user can enter different values. When the user hits the add/save button the information newly added information will be saved to the list with Java Serialization.

As stated before. If the user ends the program after adding a thing to the list, this will be saved till the next time the user runs the application.

3. The user enters the calendar:
   A view with the days during a month will appear in a scrollPane. This view also has a button for adding new things to the calendar. The user has several places to go from here. To the add view, to the help page, to the timer view or to the toDo view.

   If the user clicks on the add button a new view with empty fields appears. When adding an event to the calendar a CalendarEvent is created and added to an ArrayList with all events in the calendar. The contents of the ArrayList is then saved in Models.CalendarEvents.ser If the date of the CalendarEvent is open a FXML-component is added to the FlowPane With the corresponding dateLabel to the date that was set in the CalendarEvent. If the month is switched the FlowPanes are cleared and loaded with new dates and the CalendarEvents with matching dates are added to the flowpane the order that they were added to the ArrayList, not sorted by chronological order.

   As stated before. If the user ends the program after adding an event this event will be saved till the next time the user runs the application. The saved Events are then added back to the ArrayList of CalendarEvents when the program is started as the save file is written over by the content of the list that means that old events are not lost..

3. System design

3.1. UML-class diagram for packages

See the package UML-diagram in the APPENDIX for a reference. The application is built on the pattern MVVM as stated before and the view package and the viewmodel package is connected to each other in a few different ways. Some classes in the view package have an instance of its corresponding view model since the view wants to use information from their view model. The Interfaces package takes care of the connection between the views and viewmodels who with an instance would have a too strong dependency between each other. The ObserverInterfaces package handles the information exchange between view and viewmodel. The view observes the viewmodel and waits for new updates to display visually. The view model has instances of its corresponding model-class to get information.

3.2. Dependency analysis

See the class UML-diagram in the APPENDIX for a reference. The application is built from the different parts of the MVVM design pattern and does not contain any circular dependencies except for the observer pattern regarding the remove todo-list function. The circular dependency is between two interfaces which makes it so abstract that it can be overlooked. The different classes that represent the views do not have any dependencies to each other, they use their viewmodel to handle the information from the actions the user performs. Some of the viewmodels also have models to control the base functionality which has no direct contact with the views.

3.3. Domain model and design model

See UML-diagram for domain model and the design model (model package) in APPENDIX. There is a clear relation between these two. If you look at the main part of both UML-diagrams there is a part for the timer, a part for the todo-lists and a part for the calendar. There is also a class for the events in the calendar and the domain model has the same. The flower part and the task part from the domain model has in the application become a part of the ViewModel-package instead.

3.4. Design patterns

3.5.1. Observer pattern

A design pattern which makes it possible for multiple objects to be notified when an event happens to an object who they are observing without having multiple dependencies. This is done by implementing two interfaces, one for the observers and one for the observable. The observers classes implement the Observer interface and it’s update-method. The observable class implements the Observable interface with a notifyObservers- and register-method (you can add more methods here, for an example an remove-method). When the class who is being observed changes an object or itself it sends an update-request through the notify-method. The observers recieves this update and change their state depending on the update.

You use this pattern to get looser dependencies and to follow the Open/Closed principle since it’s easy to add new observers through the already existing method (also easy to remove observers if you have implemented such a method).

3.5.2. Factory method

A design pattern which helps with the direct object construction calls. It changes from direct object construction calls to calls to a factory method instead. When this is done the factory method returns the newly created object. You move the constructor call from one part of the program to another part. This means that you can override the factory method in subclasses and change the object that’s going to be created.

The factory method consists of an interface and this interface is common to all objects that can be produced. This interface is implemented by all the classes who wish to use this pattern. It can also consist of a class with the functionality that the classes who want to use the Factory pattern create an instance of to get access to that functionality.

3.5.3. MVVM

A design pattern which helps to build a structure of an application. It consists mainly of models, viewmodels and views. It's similar to the MVC (Model-View-Controller) but the difference is that the viewmodel does not act as a controller, it acts as a data binder between the model and the view.

In MVVM the viewmodel interacts with the model and the view interacts with the viewmodel for databinding. The view observes the viewmodel to get data and then updates and presents it. The view can also send a request to the viewmodel to either receive some data or update data.

3.5. Dynamic design

See the sequence diagrams in APPENDIX.

4. Persistent data management

4.1. Calendar

To be able to save the events the user has created when the program shuts down, the application uses Java Serialization. There is an arrayList in the CalendarViewModel class which contains all the calendar events the user wants to save. Every Time an event gets added or changed the method saveCalendarEvent is called upon. This method takes in the arrayList with all the calendar events and creates a ser-file called Models.CalendarEvents where the arrayList gets written to and therefore saved.

When the program initializes the method loadCalendarEvents in CalendarViewModel class is called upon. The method takes in a list of the FlowPanes on the Calendar page. This method reads what is in the .ser-file and creates a temporary arrayList where all the CalendarEvents, who were saved, are added. Then it calls on the addCalendarEvents method in the same class where all the calendar events get added in the public arrayList and also added to the flowPane we sent in so that the user can see them. The method loadCalendarEvent also gets called when a list gets changed.

4.2. ToDo-lists

To be able to save the lists the user has created when the program shuts down, the application uses Java Serialization. There is an arrayList in the

ToDoListViewModel class which contains all the todo-lists the user wants to save. Everytime a list gets added or changed the method saveToDoList is called upon. This method takes in the arrayList with all the todo-lists and creates a ser-file called Models.ToDoLists where the arrayList gets written to and therefore saved.

When the program initializes the method writeToDoList in ToDoListViewModel class is called upon. The method takes in a flowPane and the controller of the fxml-file where the list will be shown to the user. This method reads what is in the ser-file and creates a temporary arrayList where all the todo-lists, who were saved, are added. Then it calls on the addToDoList method in the same class where all the todo-lists get added in the public arrayList and also added to the flowPane we sent in so that the user can see them. The method writeToDoList also gets called when a list gets changed.

5. Quality

5.1 Tests

The application uses JUnit test to test the code and these can be found in the test-package.

5.2 Issues

The most obvious issue the application has is that it's strongly connected to the graphical library JavaFX. This means that when the application is downloaded and installed the user needs to have the JavaFX-library downloaded.

Another issue with the application is that STAN4J wasn't able to build package-diagrams. Therefore the package-diagrams below have been manually produced.

5.3 Analytical tools

5.3.1 Dependencies

See the UML-diagram for classes in the APPENDIX.

5.3.2 Quality tool reports PMD

The PMD plugin has been installed and run on the application and especially the problems under “design” have been looked at. The problems that are deemed critical have either been solved or improvements have been discussed.

6. References

A.Afshar. *“Understanding the Flow of Data in MVVM Architecture”*. (2019). In dev.to. Received 2020-10-16 from https://dev.to/productivebot/understanding-the-flow-of-data-in-mvvm-architecture-487#:~:text=The%20Observer%20pattern%20allows%20one,and%20keeping%20these%20layers%20separate.

Refactoring.guru, *“Factory method”,* (2020), Recieved 2020-10-20 from https://refactoring.guru/design-patterns/factory-method

Refactoring.guru, *“Observer”*, (2020), Received 2020-10-20 from https://refactoring.guru/design-patterns/observer

APPENDIX

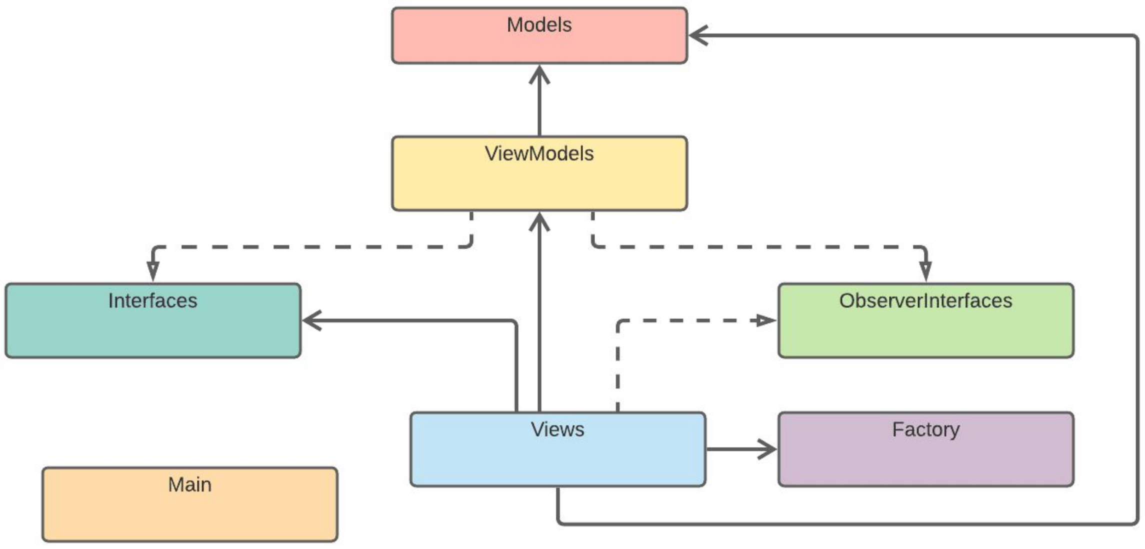

Figure 1: UML-diagram over the application's packages

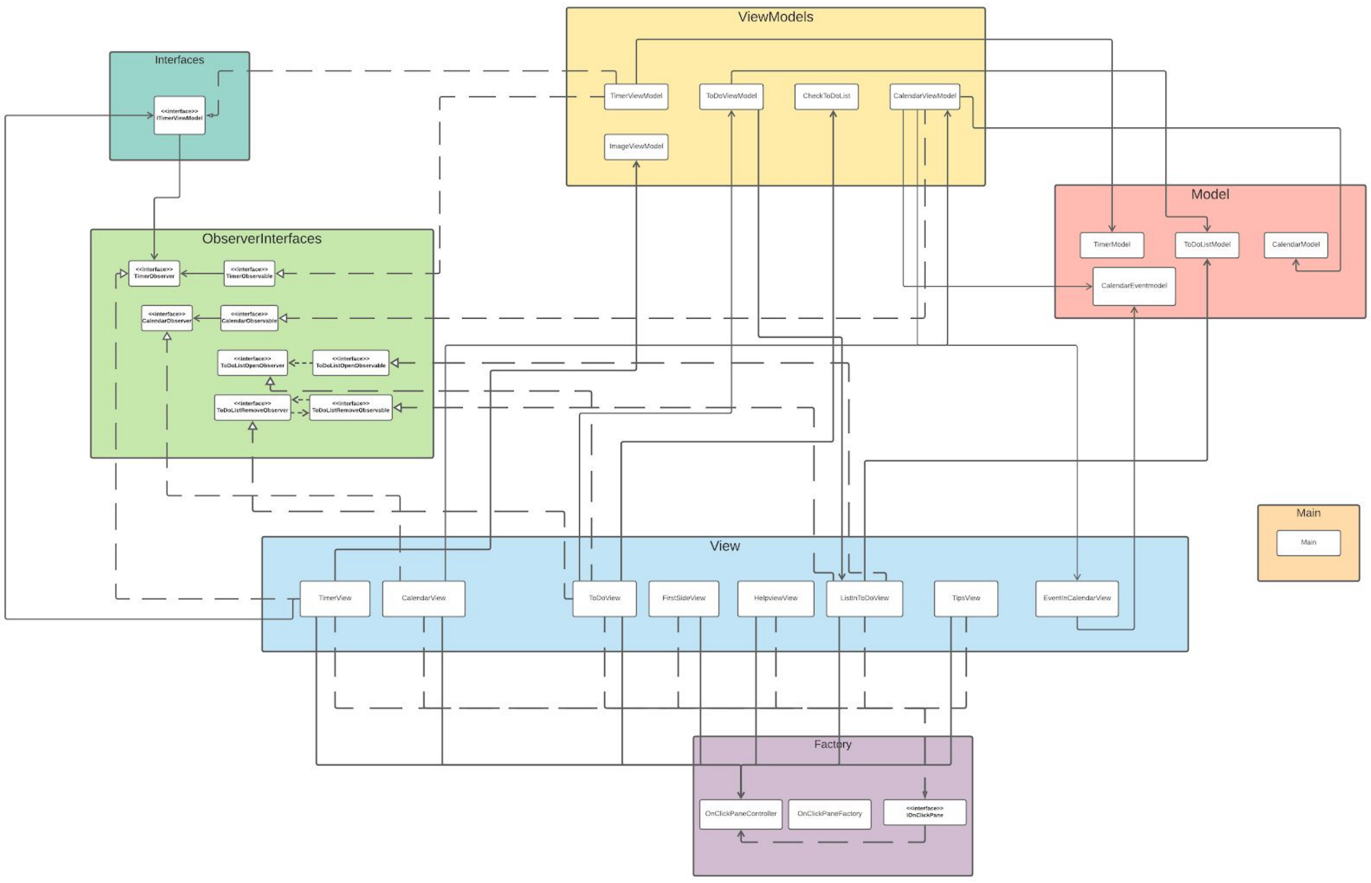

Figure 2: UML-diagram over the application's classes

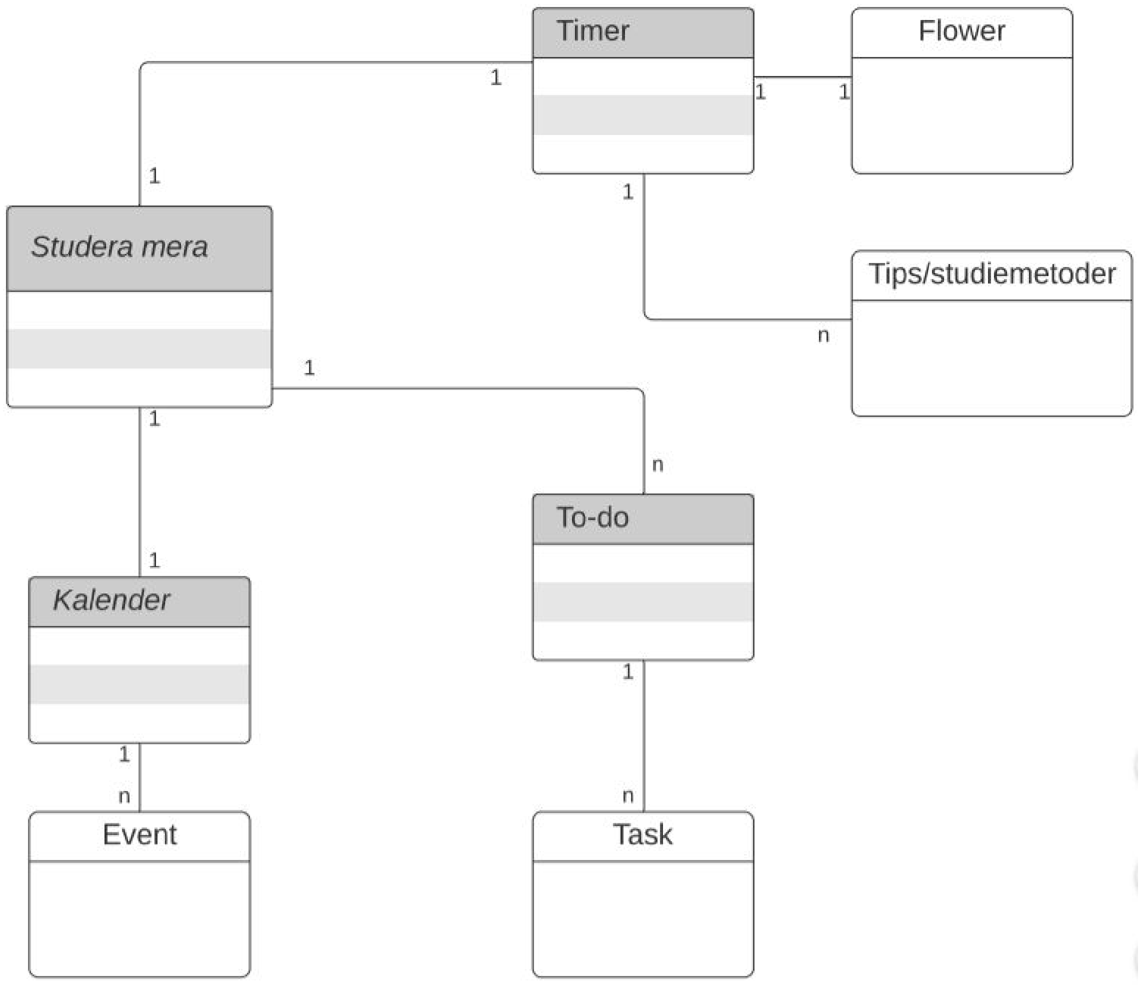

Figure 3: UML-diagram for the domain model

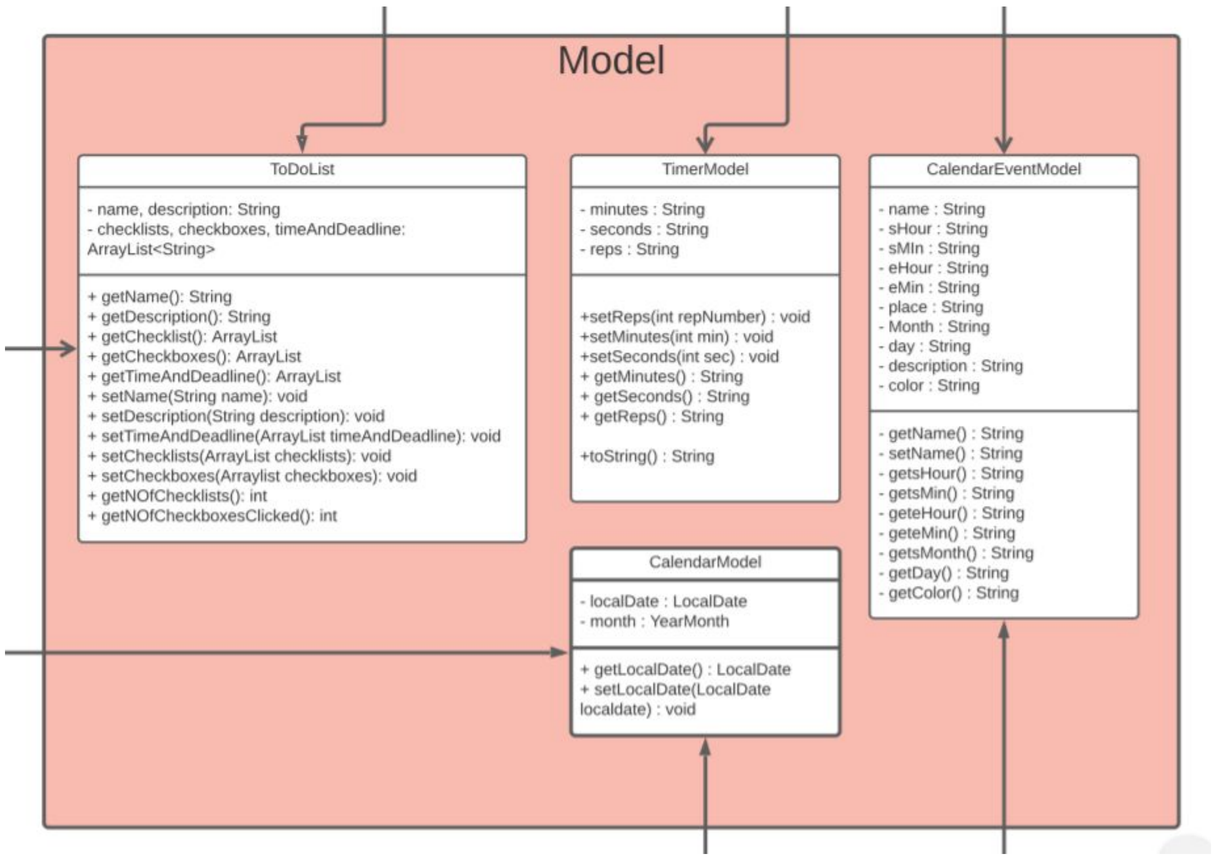
Model
ToDoList
- name, description: String
- checklists, checkboxes, timeAndDeadline: ArrayList<String>
+ getName(): String
+ getDescription(): String
+ getChecklist(): ArrayList
+ getCheckboxes(): ArrayList
+ getTimeAndDeadline(): ArrayList
+ setName(String name): void
+ setDescription(String description): void
+ setTimeAndDeadline(ArrayList timeAndDeadline): void
+ setChecklists(ArrayList checklists): void
+ setCheckboxes(Arraylist checkboxes): void
+ getNOfChecklists(): int
+ getNOfCheckboxesClicked(): int
TimerModel
- minutes : String
- seconds : String
- reps : String
+setReps(int repNumber) : void
+setMinutes(int min) : void
+setSeconds(int sec) : void
+ getMinutes() : String
+ getSeconds() : String
+ getReps() : String
+toString() : String
CalendarEventModel
- name : String
- sHour : String
- sMIn : String
- eHour : String
- eMin : String
- place : String
- Month : String
- day : String
- description : String
- color : String
- getName() : String
- setName() : String
- getsHour() : String
- getsMin() : String
- geteHour() : String
- geteMin() : String
- getsMonth() : String
- getDay() : String
- getColor() : String
CalendarModel
- localDate : LocalDate
- month : YearMonth
+ getLocalDate() : LocalDate
+ setLocalDate(LocalDate localdate) : void

Figure 4: UML-diagram over the model-package (design model)

https://lucid.app/invitations/accept/63c9ebe8-9330-4453-8738-4dfc9cce94e0

Figure 5: UML-diagram over the entire program

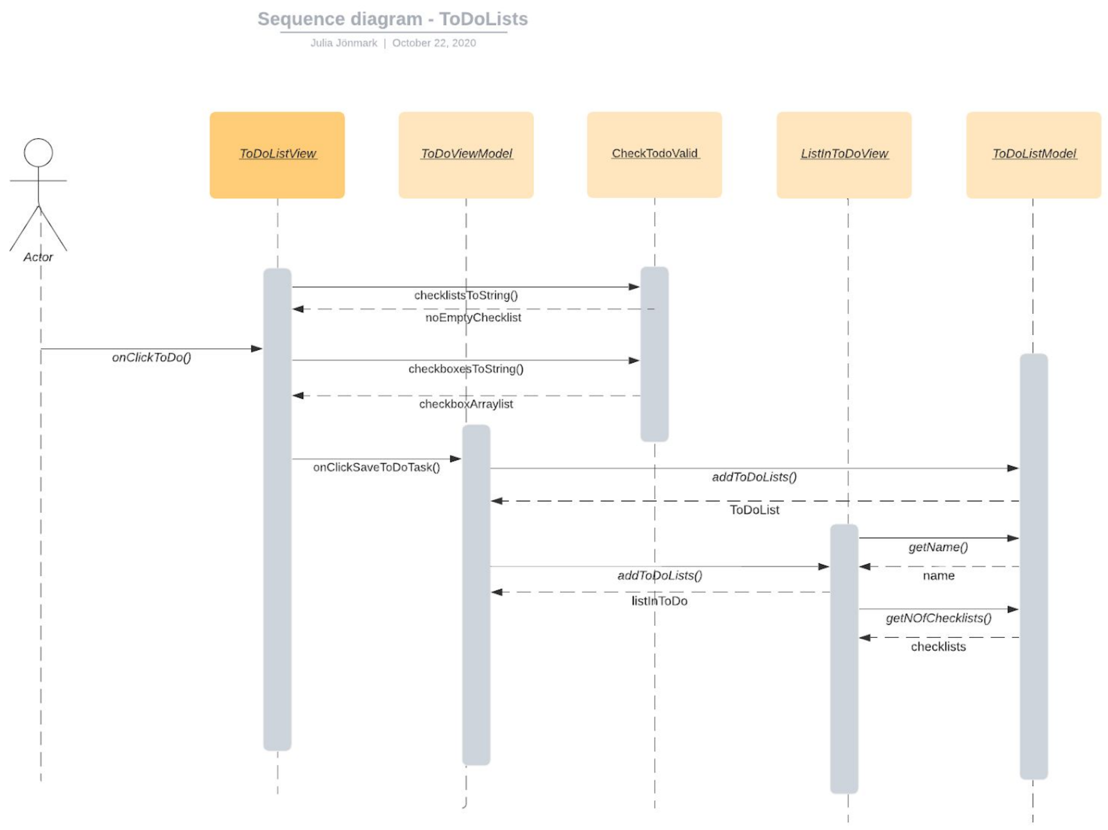

Figure 6: Sequence diagram 1 - ToDoLists

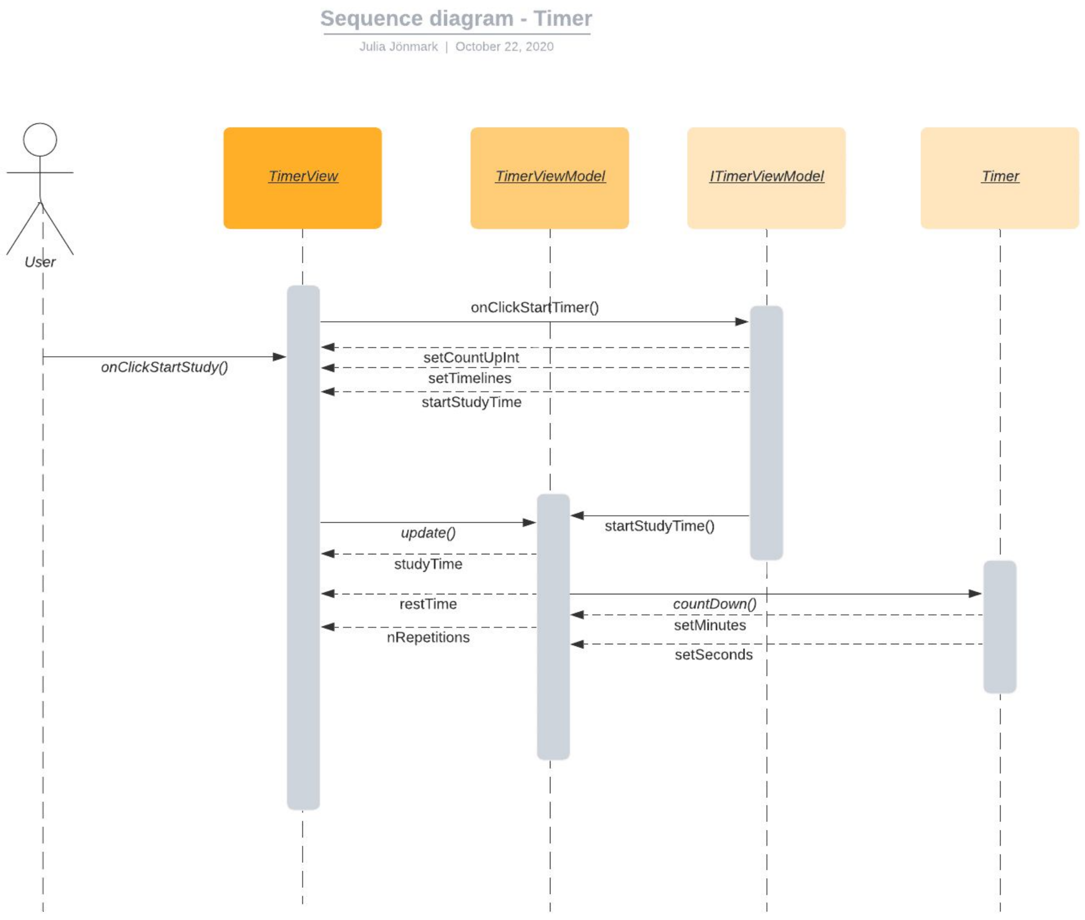

Figure 7: Sequence diagram 2 - Timer

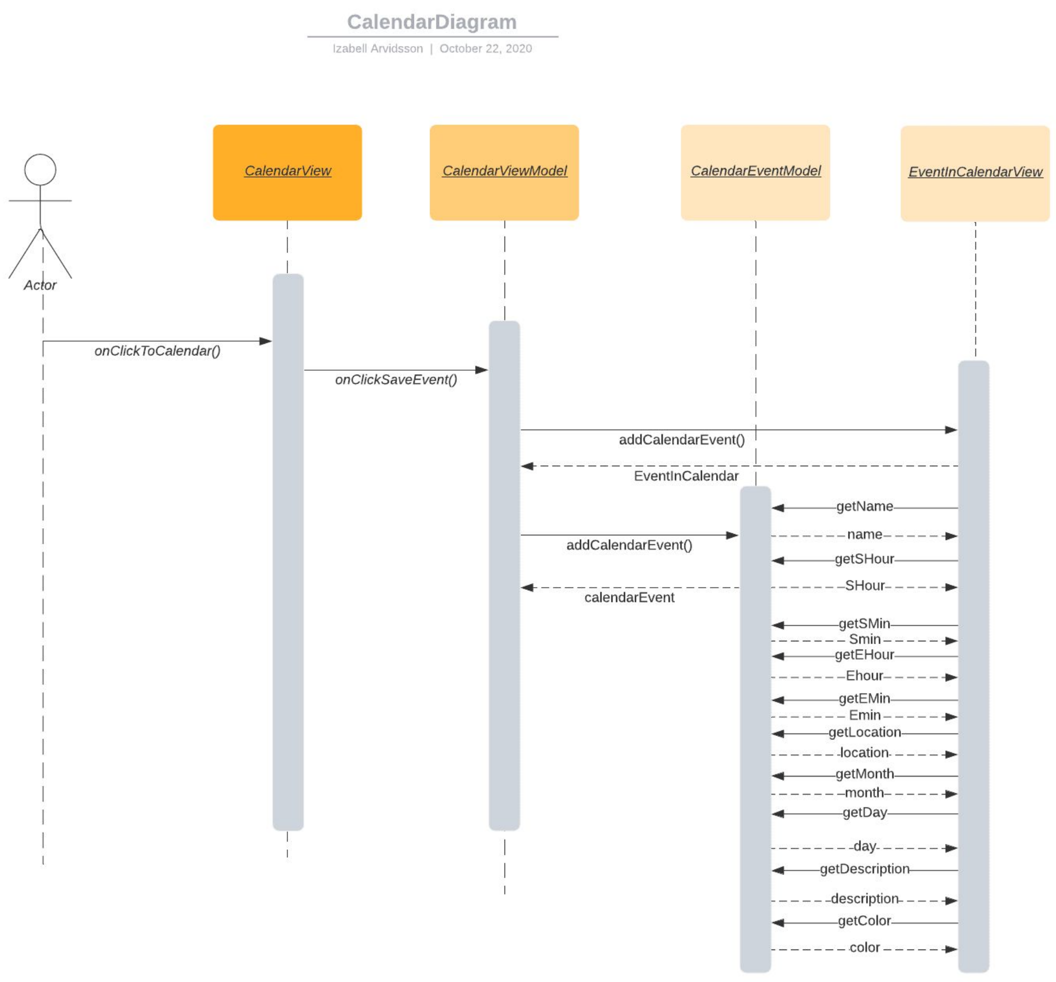

Figure 8: Sequence diagram 3 - Calendar