Final Report

StuderaMera

Date: 23/10/2020

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

This version overrides all previous versions

Contents

1. Introduction

The project aims to create a computer based application that students can use to help manage their time. The application is divided into three parts, a timer, a calendar, and a to do-list. The timer is for study sessions the user might have, where the user can decide how long the sessions will be. The calendar is for the user to add different events, making it easier to keep track of activities. The to do-list is to help the user structure different tasks that may exist. All these features to help students structure their study time.

The application is adapted to desktops and will be sized to fit most displays.

Once the application is started the user will be greeted by a first side and three options. The three choices are the three main functions in the program. The first choice is a timer. On this page the user will be able to use the timer for study sessions. The user can choose how long to study for, for how long the breaks will be and how many repetitions. The timer will count down and as the study session is progressing, a flower will grow. The flower will only grow during when the user is studying and not during rest time. If the user chooses to terminate the study session before the timer is done the flower dies. This page will also have a tips-button, where information about different study methods and techniques.

The second choice is a to do-list. On this page the user can create different lists of tasks the user has to finish. The user will be able to set a date and time for the tasks. There is also a checklist where the user can put in information, to later check the tasks of when finished. The user will also be able to delete lists that are either finished or not relevant.

The third and last choice is a calendar. On the calendar page the user will be able to see the current month, to move back and forth between months and to create events. When creating an event the user will have to put in a date, a time, and a title. The event will then be placed in the same date as the date the user put in, though all the information won't be displayed. It's when the user clicks on the event that a window pops up to show the information about the event.

1.2 Definitions, acronyms and abbreviations

Timeline - Makes it possible to update the property values along the progress of time

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

Design pattern- is a general, reusable solution to problems that often occurs in software design

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

int- stores integer type data, is primitive

Integer- stores integer types, is a class type

String- is a sequence of characters

Getter- a method that reads value of a variable

Setter- a method that updates value of a variable

Definition of Done- The acceptance criteria that are common for all user stories

ScrollPane- A function built in SceneBuilder, which this application uses to build the graphical part, that allows the application to lay tasks on each other

Checkbox- A function built in SceneBuilder, which this application uses to build the graphical part, that allows the user to check off tasks

Escape Hatch- A way for the user to get to a know place, in this application it is the first page

2. Requirements

2.1 User Stories

High Priority:

User Story
Implemented: Yes

Story Identifier: T008
Story name: Study sessions

Description
As a student who must study, I want a timer to help me concentrate better

Functional
-Will I be able to start a timer?
-Will I be able to see the timer counting down?

User Story
Implemented: Yes
Story Identifier: T007
Story name: Stop timer

Description
As a busy person I want to be able to stop the timer so if I have something else to do I can stop studying

Functional
-Can I stop the timer by clicking a button?

User Story
Implemented: Yes
Story Identifier: T001
Story name: Timer intervals

Description
As a person I want to be able to decide my own intervals for when to study and when to relax since I have difficulty focusing.

Functional
-Can I choose for how long to study?
-Can I choose for how long to rest?

-Will I be able to see how much time I have left?
-Will I be able to see a difference between study and rest time?

Non-functional:

Availability:
-Is there a limit for how long I can study?
-Is there a limit for how long I can rest?

User Story
Implemented: Yes
Story Identifier: T005
Story name: Countdown

Description
As a student I want a countdown clock so that I know if it is time to study or relax

Functional
-Can I see if it's study or rest time?
-Can I see the time ticking down?

User Story
Implemented: Yes
Story Identifier: P001
Story name: Schedule

Description
As a student I want to be able to build my own schedule so that I can easier structure my week

Functional
-Will I be able to give the events different colours?
-Will I be able to edit already existing events?
-Will I be able to delete existing events?

Security:
-Are other people able to view my schedule?

User Story
Implemented: Yes
Story Identifier: P004
Story name: Add events

Description
As a (kind of) structured person I want a button where I can add activities to my schedule so that I can have a summary over my week

Functional
-Can I add events to the schedule?

User Story
Implemented: Yes
Story Identifier: P006
Story name: Save events

Description
As a sometimes forgetful person I want to be able to save my events so that I don't forget them

Functional
-Will I be able to save an event by pressing a button?

User Story
Implemented: Yes
Story Identifier: P008
Story name: Add to do-list

Description
As a structured person I want a button where I can

Functional
-Will I be able to add tasks to to-do-lists?

User Story
Implemented: Yes
Story Identifier: P002
Story name: To do-list

Description
As a student I want to be able to make a to do-list so that I don't miss something

Functional
-Will I be able to edit existing tasks?
-Will I be able to see an overview of the task?
-Will I be able to check off tasks?

Security:
-Are other people able to view my to-do-lists?

User Story
Implemented: Yes
Story Identifier: P007
Story name: Save tasks

Description
As a sometimes forgetful person I want to be able to save my tasks so that I don't forget what I have to do for the day/week

Functional
-Can I save my tasks by pressing a button?

Medium Priority:

User Story
Implemented:
Story Identifier: P009
Story name: Edit tasks

Description
As a person I want to be able to edit already existing tasks

Functional
-Will I be able to edit tasks that are already saved?

Non-functional:
Availability:
-Will there be a button that indicates that the task is editable?

User Story
Story Identifier: P010
Story name: Edit events

Description
As a person I want to be able to edit events that are save to the calendar so that if an event changes I can adjust the event

Functional
-Can I edit events after they are saved?

User Story
Implemented: No, due to not enough time
Story Identifier: P003
Story name: Overview for calendar

Description
As a student I would like to see an overview over what my schedule looks like

Functional
-Can I see an overview of my day?
-Can I delete events from the overview?
-Can I edit events from overview?

Security:
-Are other people able to see my overview?

User Story
Implemented: No, due to not enough time
Story Identifier: P005
Story name: Overview for to do-list

Description
As a student I want an overview over the tasks I must do during the week or day

Functional
-Will I be able to see an overview the different to do-lists
-Can I delete to do-lists via the overview?
-Can I check off the tasks?

User Story
Implemented: Yes
Story Identifier: A002
Story name: Escape Hatch

Description
As a person who is not very technical, I want a button that easy takes me back to the front page

Functional
-Will I be able to always find my way back to the first page?
-Will the changes I made be saved if I leave the page?

Non-functional:
Availability:
-Will the escape hatch be available everywhere?

User Story
Implemented: Yes
Story Identifier: T006
Story name: Booster

Description
As a somewhat unmotivated student I want to get some kind of visual booster to keep studying

Functional
-Will the visual booster be visual on the screen at all times?

Non-functional:
Availability:
-Can I leave my study session without any consequences regarding the booster?

User Story
Implemented: Yes
Story Identifier: T004
Story name: Motivation

Description
As a person who often leaves applications, I want something visual that makes me stay

Functional
-Can I do something else in the application while the timer is ongoing?

Non-functional:
Availability:
-Will the calendar be available?
-Will the to do-lists be available?

User Story
Implemented: Yes
Story Identifier: T002
Story name: Set intervals

Description
As a student I want to be able to select my own intervals for studying since it will help me get more motivated

Functional
-Can I choose a specific time to study and rest for?
-Can I see how much I have left of the interval?

Non-functional:
Availability:
- Can I set an uneven number for both study and rest time?

Low Priority:

User Story
Implemented: Yes
Story Identifier: T003
Story name: Tips and techniques

Description
As a person how wants help with staying focused, I want to get tips and facts about different methods how to keep focus since I want to be more effective

Functional
-Can I find help on how to better my studying techniques?

Non-functional:
Availability:
-Will I be able to add my own methods?
-Will I be able to find the sources?

User Story
Implemented: Yes
Story Identifier: A001
Story name: Help

Description
As a person I want a help button where I can go if I get stuck

Functional
-Can I get help if I get stuck?

Non-functional:
Availability:
-Can I access the help page from anywhere?

User Story
Implemented: Yes
Story Identifier: A003
Story name: Beginner

Description
As a beginner of the application I want a tool that can help me if need be

Functional
-Will I easily see where I can go?
-Will there be a walkthrough of the application?

Non-functional:
Availability:
-Will I be able to get help from anywhere in the application?

2.2 Definition of Done

Not only must the user stories meet the acceptance criterion for just that story, but they must also meet the mutual acceptance criterion. The common acceptance criterion we have used for these user stories are:

-They should be tested
-They should be reviewed

2.3 User interface

The user interface is created to have a consistent design throughout the whole application. This is made so to make the user feel more comfortable and to faster get to know the applications and its features. Once the user starts the program the first page will appear with three choices. The three different buttons lead to either the timer, the to-do-list, or the calendar.

Figure 1: First Page

2.3.1 Timer

By clicking on the “Till timer”-button the user will be sent to the timer page. On this page the user will be greeted by three different spinners, a “Starta”-button, and a button with a light bulb. With the spinners help the user can decide for how long to study or rest. The user does so by clicking on the tiny arrows that the spinners have. There will be a limit for how long to study, which is 100 minutes. The same goes for rest time, where the limit is 40 minutes. Another limit that is relevant to both times is that the spinners takes 5 minutes in one step. This means the user will only be able to choose a time that ends in either 0 or 5. There is a third spinner, and this on is for repetitions. Here the user can decide for how many times to repeat the study and rest time.

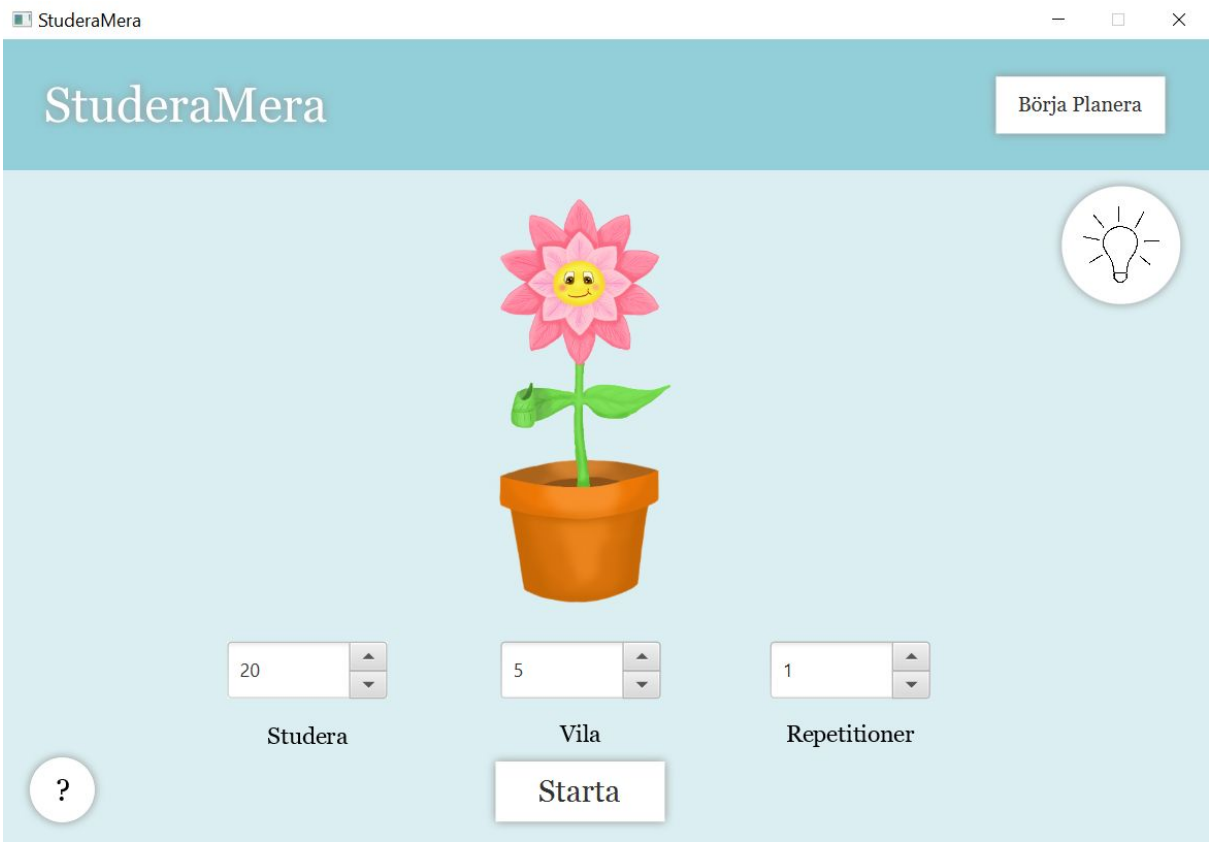

Figure 2: Start page for the timer

Once the user has chosen a time for both study and rest, all the user must do to start the timer is click the “Starta”-button. Once the button is pressed the user will be taken to a new page and the timer will start to countdown. During the study period an image of a flower will appear on the screen. The flower will grow successively with the time chosen for study and the number of repetitions. The application is coded so that the flower will not grow during rest time.

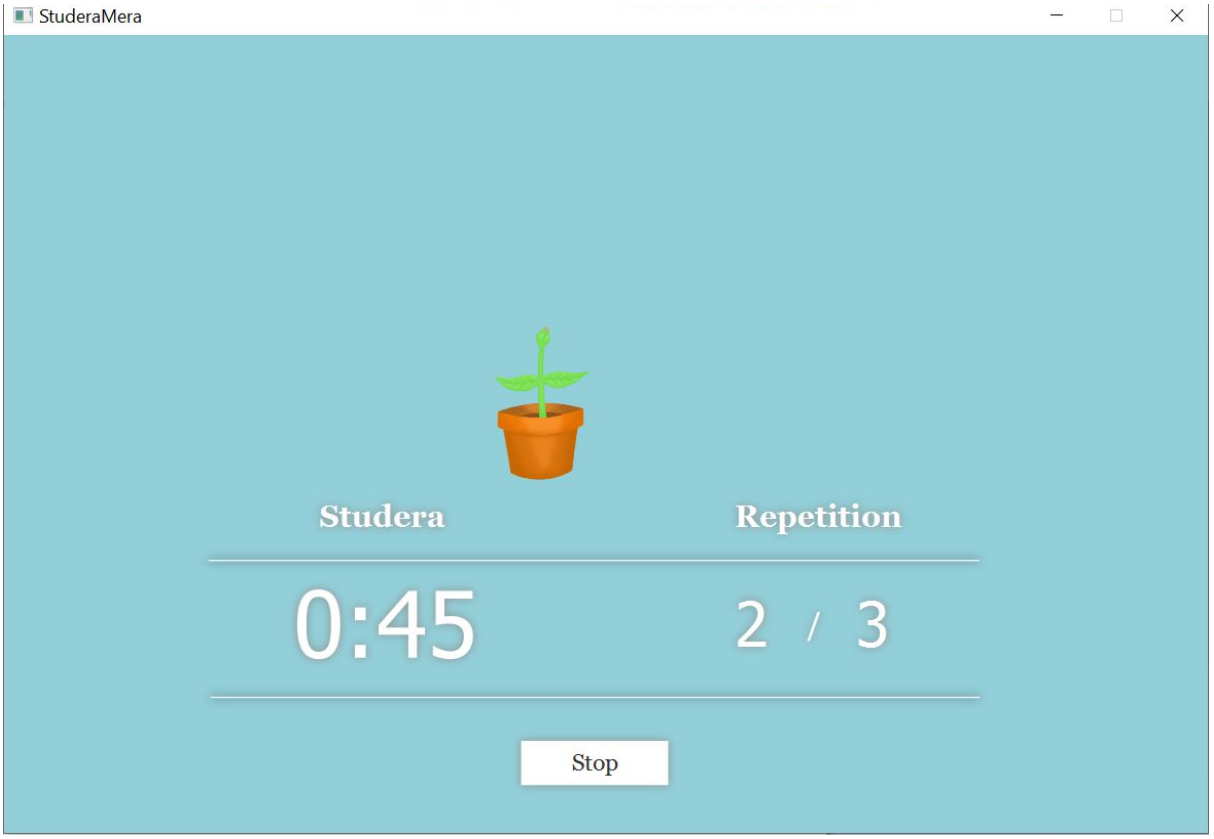

Figure 3: The timer is running

When the study period is completed a new window appears to show the user that the session is done. The meaning of this is go give the user a feeling accomplishment. Once the “Tillbaka till startsidan”-button is pressed the user will be taken back to the timer page.

Figure 4: The study session is done

If the user wishes to end the session before the time has run out, the user can do so by pressing the “Stop”-button. The countdown will then be paused, and a window will appear, asking the user if the session is to be ended. If pressed no, the user will go back to the timer and the countdown begins again. If pressed yes, a window will appear with a dead flower and a message that lets the user know they failed the study session. The message also reminds the user to think of a good time to study for, especially if the time chosen was not working.

Figure 5: The user has decided to end the study session before time was out

By pressing the “Jag förstår”-button the user will be taken back to the timer page. Here the user can decide on a new time or go to the calendar or to do-list.

2.3.2 To do-list

If the user chooses to go to the to-do-list, the user will be met by an overview of the to-do-list. Here the user can add, change, and delete tasks.

Figure 6: The to do-list page

To add a task the user must press “+”-button. A window will then pop up where the user can write specifics for just that task. The task can have a title, a description, and a checklist. The user will be required to fill in a deadline and a time in which the task will be completed, to be able to add the task.

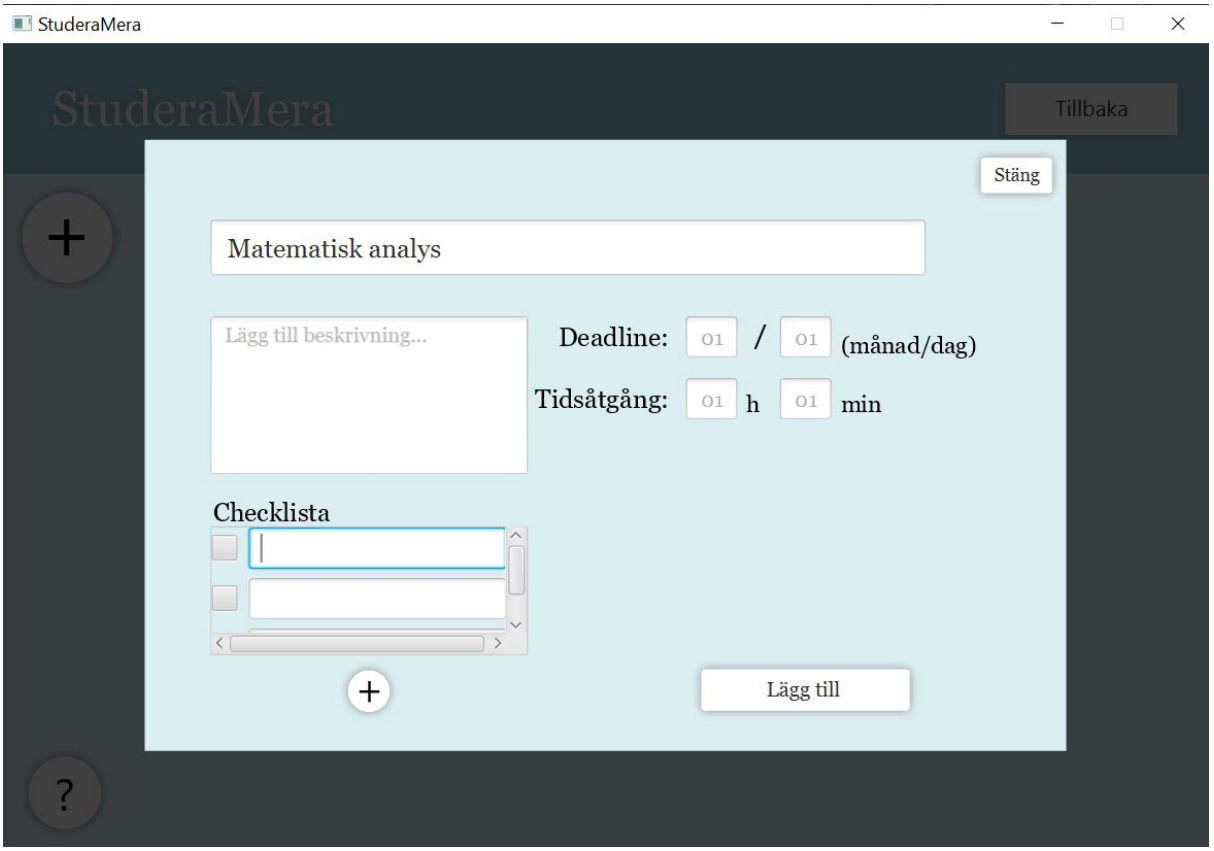

Figure 7: Adding a task

After all fields are filled in the user must press “Lägg till” and the task is saved. The task will then be seen in the grey scrollpane. If the user by chance wrote wrong or the task itself changed the user can change the task by clicking on it.

Figure 8: The task is added to the to do-list

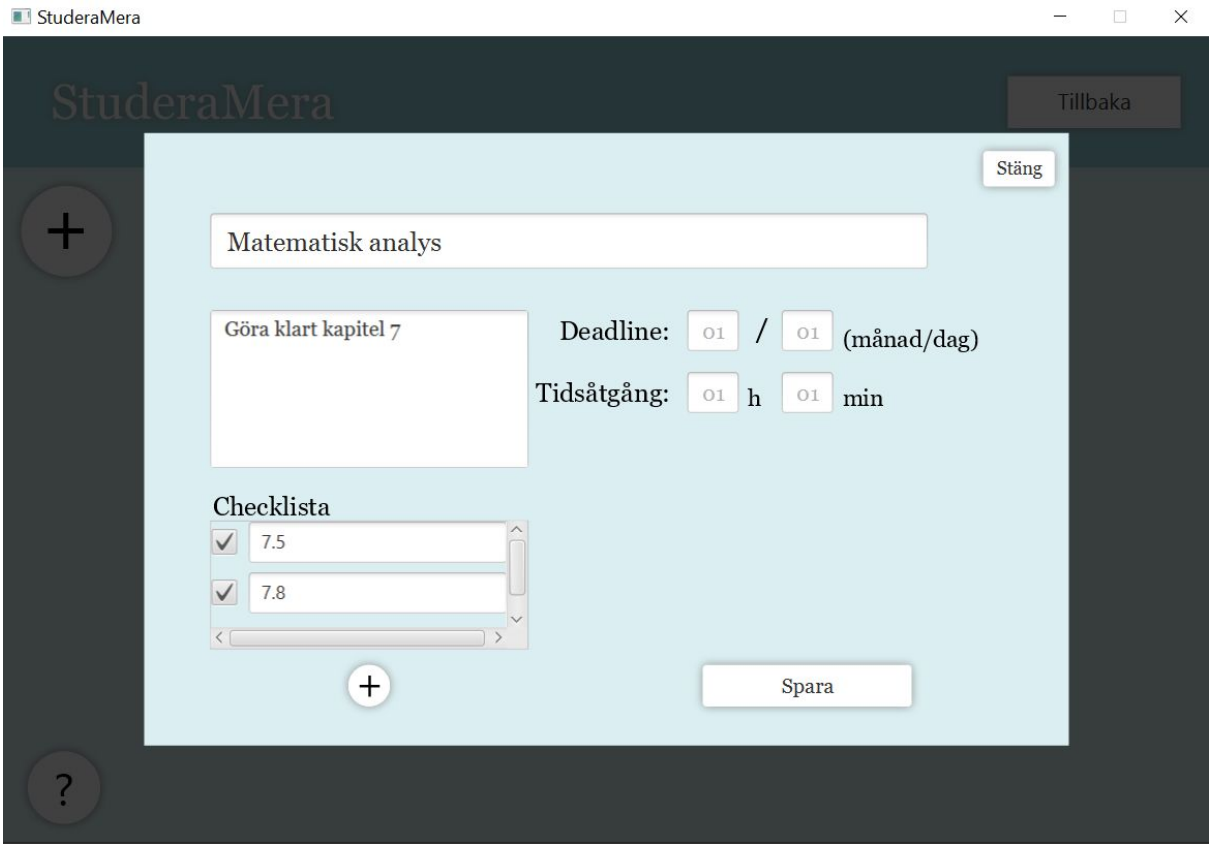

Figure 9: Changes are made to a task

In the overview of the to-do-list page the user can see the different tasks that have to be completed. On the task there is information regarding how many of the task's checklists are done and if the whole task is completed. There is also an image of a trash can. By clicking on the image the task will be deleted. The user can check off the whole task by clicking on the grey checkbox and uncheck it by clicking on the checkbox again.

Figure 10: A task is checked off

2.3.3 Calendar

By clicking on the “Till kalender”-button the user is taken to a page that has an overview of the current month. The current day has a different colour then the other days, to indicate that is the day.

Figure 11: The calendar page

Here the user can add events to certain days by pressing on the “+”-button. The user can then give the event a title, a description, a location and a colour. The event requires a date so that it can be placed on the same date that the user decided. The event is saved once the user clicks on the “Lägg till”-button.

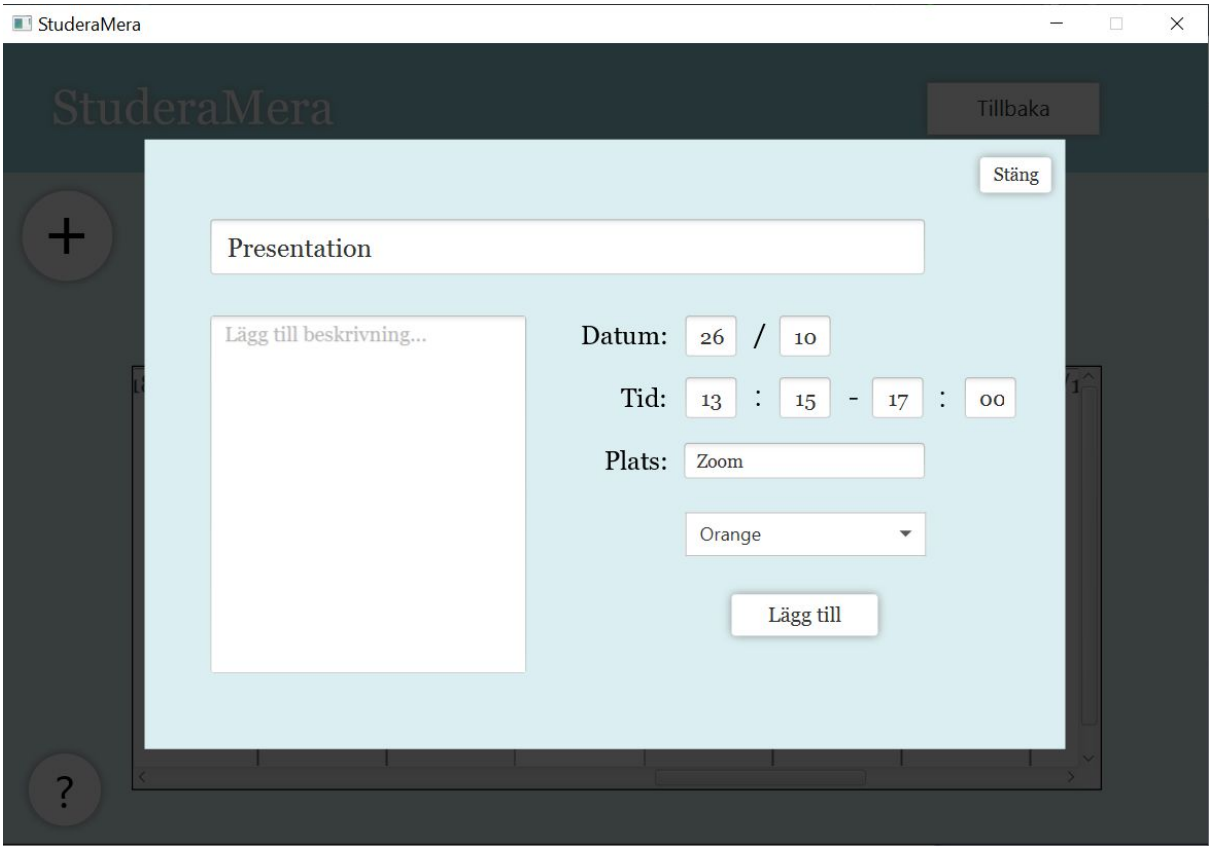

Figure 12: Adding an event

Figure 13: The event is added to the calendar

If pressed the “Föregående”-button changes the overview to the month before. There is not a limit for how long back the user can go. The same applies to the “Nästa”-button, which changes to the next month. To make the user aware of the change, the date under the header “Din kalender” changes to the corresponding month.

2.3.4 Help page

The help page can be reached from every page, except when the timer is running. By pressing the button with a question mark, in the lower left corner, the user will be transferred to the help page. Here the user can get information on often asked questions, contact and references. From here the user can go back to the first page to then decide what to do.

Figure 14: The help button is circled in red

2.3.5 Back button

On every page the user can find a “Tillbaka”-button. This button will take the user back to the first side, where the user will have three choices.

Figure 15: The back button is marked in red

3. Domain model

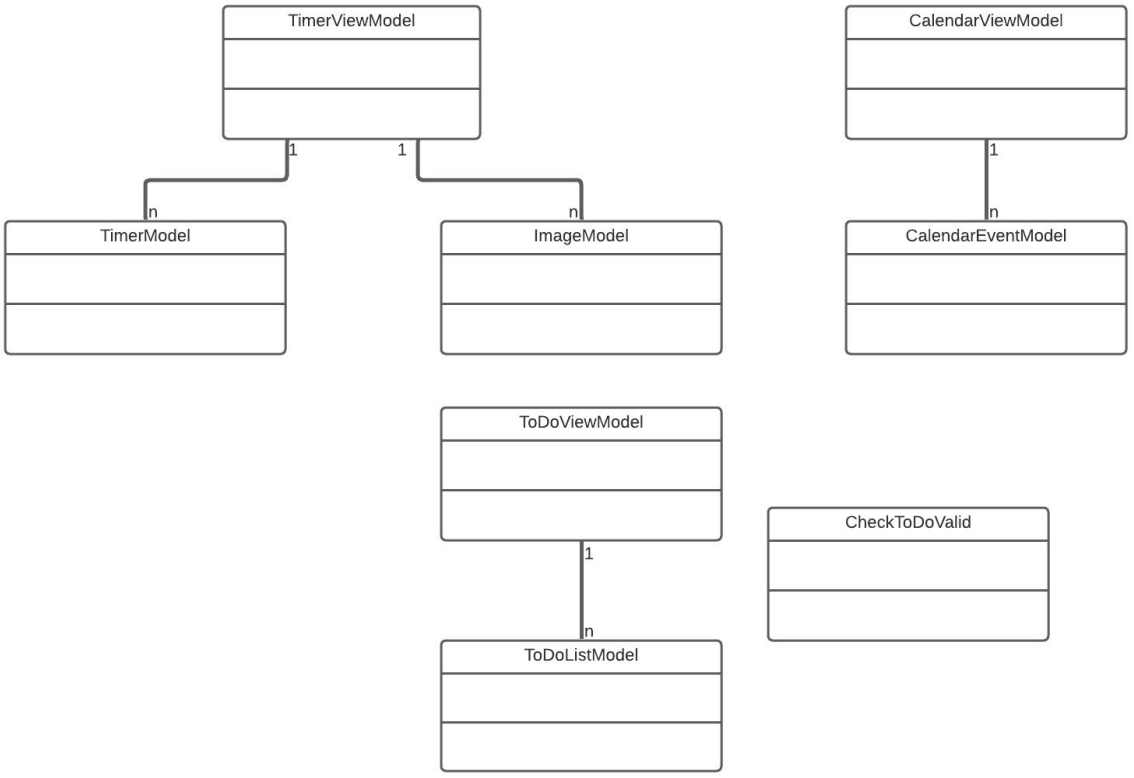

Figure 16: domain diagram

3.1 Class responsibility

CalendarEventModel

This class holds the constructor for a calendarEvent as well as necessary setters and getters.

CalendarViewModel

This class is responsible for the logic behind saving calendar events while the program is running and between startups. It also makes sure that the calendarEvents are displayed and in the correct FlowPane.

CheckToDoValid

This class is not dependent on other classes. Its responsibility is to check the status of the checklists and checkboxes it gets from the ToDoListView class in its methods.

ImageViewModel

This is the model that is responsible for sending the right number for the image that the TimerView is supposed to show. Takes in the amount of studyseconds that has passed and the total studytime in seconds and returns an int that the TimerView receives.

TimerModel

The model that is responsible for the converting of the time in form of integer to a string which will be shown by the TimerView.

TimerViewModel

Handles the functionality of the timer and creates timelines which will be going through the class' methods whilst running. It sends updates about all the different parts connected to the timer to the TimerView, through the Observer pattern, which updates the graphics.

ToDoListModel

The ToDoListModel-class creates a new todo-list with the data it got from the user from the ToDoViewModel. A todo-list contains a name, description, timeline, deadline and checklists. By calling on the class getters or setters the values can be reached and changed.

ToDoViewModel

This class toDoLists handles the functionality regarding the todo-lists. It has a list of all the todo-lists that it has created and saves them when the program shuts down and writes them back up when the program starts again. It has dependencies on ToDoListModel and the ListInToDoView because the class creates an instance of them.

4. References

Oracle.com, “ScrollPane”, (2020), Recived 2020-10-21 from
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ScrollPane.html

Oracle.com, “Timeline”, (2020), Recived 2020-10-21 from
https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Timeline.html

Codejava.net, “Java Getter and Setter Tutorial”, (2019), Recived 2020-10-21 from
https://www.codejava.net/coding/java-getter-and-setter-tutorial-from-basics-to-best-practices

Tutorialspoint.com, “Difference Between an Integer And Int in Java”, (2019), Recived 2020-10-21 from
https://www.tutorialspoint.com/difference-between-an-integer-and-int-in-java

Wikipedia.org, “Software Design Pattern”, (2020), Recived 2020-10-21 from
https://en.wikipedia.org/wiki/Software_design_pattern

Medium.com, “The Definition of Done, What Does ‘done’ Actually Mean?”, (2017), Recived 2020-10-21 from
https://medium.com/@dannysmith/the-definition-of-done-what-does-done-actually-mean-ef1e5520e153

Peer review of group 10

Do the design and implementations follow any design principles?

Looking through the code you will see it follows the Single Responsibility

Principle with how the classes are built, very class taking care of one thing. The code also follows the Liskov Substitution Principle, given how the classes Notebook and Book extends Item.

Consistent coding style?

The code style is consistent since it uses the MVC pattern. Since the code is divided like it should be, the model-classes contain the same type of code and the controller-classes contain the same type of code.

Is the code documented?

Yes, for the most part, especially for the methods. There are comments in all the classes, except MainActivity and MassagePage. Would like to see more comments about the classes, like what they do.

Are proper names used?

Looking through the package there are three classes, whose names may not give away exactly what they do. The classes are ListingAdapter, ListingPageActivity and Listing. But after looking into the different classes and looking at the methods you get a grip of what they do.

The names of the methods and variables are good and easy to understand.

Is the design modular?

The code is to some extent modular. The code has no circular dependencies , since the code is divided into smaller classes that does one thing. There is also no unnecessary dependency in the code, though there are some strong ones. Both Book and Notebook extend an abstract class Item. Having looked at the code, the dependency makes sense.

Is the code well tested?

The model package is well tested, but since the group writes themselves in their SDD that there is some functionality in the ControllerView package, it should also be tested.

Are there any security problems?

One security problem that was found were that if the user were to write two different passwords in the register page, the user would still get logged in.

We also tried sending in an ad without a picture, which made the program exits and the ad was not saved.

Is the code easy to understand?

The code is easy to understand since its names of methods, classes, and variables. The classes are relatively small which makes it easy to understand what the class are supposed to do and therefore also how the methods work. Since the code follows the MVC pattern it is also easy to understand what the different classes are responsible for.

Does the code have a MVC structure?

Yes, somewhat. The Model is separated from the View and Controller. The controllers and views are in the same package but seeing how the code is structured it makes sense to have them in the same package.

Can the design or code be improved?

The design is very easy on the eye and the code looks good, so no.

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