

# Exam Project 2021



**Handed in by:**

Kamila Potasiak

László Soma Tolnay

Jakub Czechowski

## **Preface**

This report is a part of the 2<sup>nd</sup> semester's exam project at Business Academy Southwest. It introduces, explains, and summarizes the strategic choices, implementation and teamwork process, and final solutions that have been made to create a desktop application for Arla Esbjerg. A desktop application has been requested by the company Arla, to be able to store various forms of dairy production-related data (KPIs) and allow the workers to follow this data during their work. Since more screens are involved in the production process, the aim of the project is to provide a software solution where the KPI Management System's data would be presented on the screens efficiently without using numerous monitors. All essential information which is required to run the application properly is included in Appendix D.

## Table of Contents

<b>1. Introduction.....</b>	<b>5</b>
<b>1.1. Background .....</b>	<b>5</b>
<b>1.2. Problem statement .....</b>	<b>5</b>
<b>1.3. Report structure.....</b>	<b>6</b>
<b>1.4. Product vision .....</b>	<b>6</b>
<b>1.5. Strategic analysis .....</b>	<b>7</b>
<b>2. Pre-game.....</b>	<b>11</b>
<b>2.1. Project organization.....</b>	<b>11</b>
<b>2.2. Overall project schedule .....</b>	<b>11</b>
<b>2.3. Initial product backlog .....</b>	<b>12</b>
<b>2.4. Architecture .....</b>	<b>12</b>
<b>3. Sprint 1.....</b>	<b>14</b>
<b>3.1. Sprint planning .....</b>	<b>14</b>
<b>3.2. GUI (including UI-design patterns).....</b>	<b>14</b>
<b>3.3. Data model.....</b>	<b>19</b>
<b>3.4. Implementation .....</b>	<b>20</b>
<b>3.4.1. Code examples.....</b>	<b>24</b>
<b>3.4.2. Design Patterns/principles .....</b>	<b>26</b>
<b>3.5. Sprint Review.....</b>	<b>28</b>
<b>3.6. Sprint Retrospective .....</b>	<b>28</b>
<b>4. Sprint 2.....</b>	<b>29</b>
<b>4.1. Sprint planning .....</b>	<b>29</b>
<b>4.2. GUI (including UI-design patterns).....</b>	<b>31</b>
<b>4.3. Data model.....</b>	<b>34</b>
<b>4.4. Implementation .....</b>	<b>36</b>
<b>4.4.2. Design Patterns/principles .....</b>	<b>44</b>
<b>4.4.3. Unit test.....</b>	<b>46</b>
<b>4.5. Sprint Review.....</b>	<b>47</b>
<b>4.6. Sprint Retrospective .....</b>	<b>47</b>
<b>5. Conclusion .....</b>	<b>47</b>
<b>6. References.....</b>	<b>48</b>
<b>7. Appendices .....</b>	<b>49</b>
APPENDIX A .....	49

APPENDIX B.....	50
APPENDIX C.....	51
APPENDIX D .....	52
APPENDIX E.....	53

# 1. Introduction

## 1.1. Background

Arla Foods, the partner of this project, is a Danish farmer-owned multinational cooperative which is worldwide by its numerous products in the dairy sector. Based on their website, with around 19. 000 employees internationally, Arla is one of the 4<sup>th</sup> biggest dairy producer company in the world. The headquarter is in Viby, Denmark but the production has several locations across the globe.<sup>1</sup> The current project is focusing on two locations Esbjerg Dairy and Cocio, these two are usually referred to as Esbjerg Dairy Center in daily corporate communication. Esbjerg Dairy Center employs approximately 400 employees and produces several products for big brands such as Mathilde, Starbucks and Cocio products.

According to Bhatti et. al (2014), when a company wants to stay profitable and to monitor the production processes in a balanced way, the so-called **key performance indicators** (from now on: **KPIs**) provide the greatest help. Once an organization has defined its own mission, vision and goals, it is necessary to be able to measure the effort and progress made to achieve the goals. The goal of the KPIs in dairy production management is to assign a measurable value to how efficiently Arla's farms perform and able to achieve their goals as manufactures. KPIs are also useful because they are a good communication tool both inside and outside the company, as KPIs can also compress more complex information. KPIs provide feedback in an understandable, clear and fast way that is easy to process and use in everyday life when it comes to operators in the production. In the case of Esbjerg Dairy Center, the tools for KPIs are owned by the company and shown on several monitors for the participants who are involved in the production.

However, they want to change this system, which will also increase efficiency in the future through better transparency and simplified solutions.

## 1.2. Problem statement

For practical reasons (saving resources, time management, etc.), the client -Esbjerg Dairy Center- wishes to reduce the number of monitors in the future by introducing larger screens where the KPIs are displayed divided into dynamic grids separately. Therefore, **the project's goal is to create a system that is capable of multi-screen splitting, allowing different sources to be displayed.** The project period was limited to 3 weeks in total, so the time constraint was a governing factor throughout the working process and implementation.

The following limitations were concluded by the developer team based on the client's introductory presentation from the kick-off meeting and used as guidelines for the project:

### **Standards of behavior:**

As the first basic guideline, the application must allow for the users to be able to choose between multiple screens, which the given user has specific access to. It is required, that the screen must be built from dynamic grids which could be controlled in the administrative module by the admin(s). It is important that the grids can display different sources, namely

---

<sup>1</sup> <https://www.arla.dk/om-arla/landmandsejet/>

HTTP, PDF, Image, Video, Excel, RDP and so on. Plus, the sources must be zoomable and needs to be refreshed. Therefore, the administrative module should be able to update the screens based on a certain time interval and configure them also.

#### **Administrative module's criteria:**

Inside the administrative module, the admin(s) must manage the users (editing, adding, deleting) and gives permission for a user could be assigned to multiple set of screens. For securing privacy, the admin(s) must have an individual and secured username with a password for the login. It is also required, that the admin(s) must be able to see through every screen in use with their date displayed too; and could make individual changes regarding the layout of the screens.

#### **Other technical requirements:**

The system's all configurations must be stored in an MSSQL server which is a major technical requirement set by the clients. In addition to the MSSQL server, the program must be available offline.

### 1.3. Report structure

The report's first chapter starts with a brief **background information**, which was followed by the definition of the problem areas (see **Problem statement**) with limitations. Then certain academic tools (concepts, methods, etc.) were chosen to describe the product vision, the role of the stakeholders and get an overview of the risk management. The second chapter, '**Pre-Game**' explains the project organisation (projects schedule, working agreement, initial product backlog, program architecture). The next chapter is about **Sprint1**, where the backbone of the application was settled right after the kick-off meeting and later during the daily meetings. The structure of the program was determined (GUI) and planning the design of the database was stated. This was followed by the implementation of the dreamed prototype in practice. The work which was done so far was presented to the clients and they shared their feedback too at the first sprint review meeting. After the meeting, a sprint retrospective was conducted by the developer team, where an overall look was taken into the findings so far and to the possible future steps. The fourth chapter, '**Sprint2**'; was based on the data and experiences gathered during Sprint1 and based on that, the purpose was to develop better implementations in the program. It was closed with another sprint review meeting with the stakeholders and a sprint retrospective. Finally, in the **conclusion**, the determining problem areas were restated, then the used decisions for the program implementation were evaluated and summarized.

### 1.4. Product vision

Simply put, Product Vision is an easy and great tool to determine the product's overall purpose by explaining the intentions for the creation and what actual standards should the product meet to fulfil the target group's (stakeholders & users) requirements. Based on Scrum.org, the Product Owner (in short PO) is the actual owner of the product which was implemented by the Development Team (DT). During the project's working process, the DT

mastered together the product vision according to the established requirements by Esbjerg Dairy Center (the PO). It was frequently rediscussed and readapted after the Scrum review and retrospective meetings to ensure the best solution implementations which must meet with the PO's expectations and needs.

In order to decrease the number of used monitors in the daily production process and let the employees (as the main users) to find the requested data (KPIs) easier, an implementation of a special desktop application was asked by the company. Using the provided information, Pichler's template (2014) and the elevator pitch technique; the following product vision were formed by the DT:

*Our product is visioned for all the employees who participate in the production process (users) and IT department employees (admins) working at Arla's Esbjerg Dairy Center. Employees as product users would have the opportunity to follow the production related KPIs on multiple screens simultaneously in a well-organised transparent way and accessed employees as admin(s) could make changes in the stored data/user administration in the program among other functions. The KPI Management System is a desktop application, which is specially implemented for internal use in dairy production. It is a unique product because it allows to display information offline on multiple screens in different kind of sources on one monitor without using several monitors in the production. That will lead to decreased maintenance costs and time waste. Moreover, it will increase productivity and efficiency too. Unlike other similar competitive alternatives on the market, the KPI Management System is easy to use as a user and easy to manage as an admin. The other additional benefits for the clients the product can be well developed and expanded in the future. In this way, a possible future collaboration between the consumers and the developers will be mutually beneficial opportunity.* The complete Product Vision Board template where the details were listed and explained by the development team can be seen in Appendix A.

## 1.5. Strategic analysis

After the settlement of the Product Vision, which helps the team to understand the targeted goal and move forward to reach it; as a next step, a brief strategical analysis was conducted. Planning of strategical processes involves a certain set of actions that were taken to achieve the envisioned result which was described in the Product Vision. The above-used information was given by the client and collected by the DT. This information forms the backbone of the strategy construction which could be divided into a stakeholder analysis and into a risk analysis & management.

### **Stakeholders**

A stakeholder can be a person, group or organization that has an interest or concern regarding a certain product/project. Stakeholders can influence or be influenced by the actions of the organizations. The most important stakeholder of this project is the PO who is the legal owner of this program. In the case of many stakeholders in one project, self-interest could be a problem in the long term. Conflicts between the parties might occur especially when the time required to develop the project exceeds the allotted time. During the project period, some previous requirements (which were presumably made by the client) were removed by the

teachers. For instance, the screen resolutions in pixels and centimetres became not part of the exam anymore.

### Stakeholder analysis

The picture below shows the project's stakeholders. The team jointly decided how much power and interest each individual stakeholder has. The reason why end-users are yellow because employees might require additional persuading to change for this program in the production. This assumption rooted in human nature, namely, after getting used to a kind of system, it takes time and energy to get used to a new one. Since the DT had zero contact with the employees at Esbjerg Dairy Center, it is just an objective assumption and difficult to manage as a risk. The school (EASV) is coloured yellow because of its neutral role as an observer in the project. However, the rest of the unmentioned stakeholders are identified as advocates/supporters of the program (green colour), they still must be constantly managed. This is especially valid for the product owner because obviously the highest interest and power in the program belongs to there.

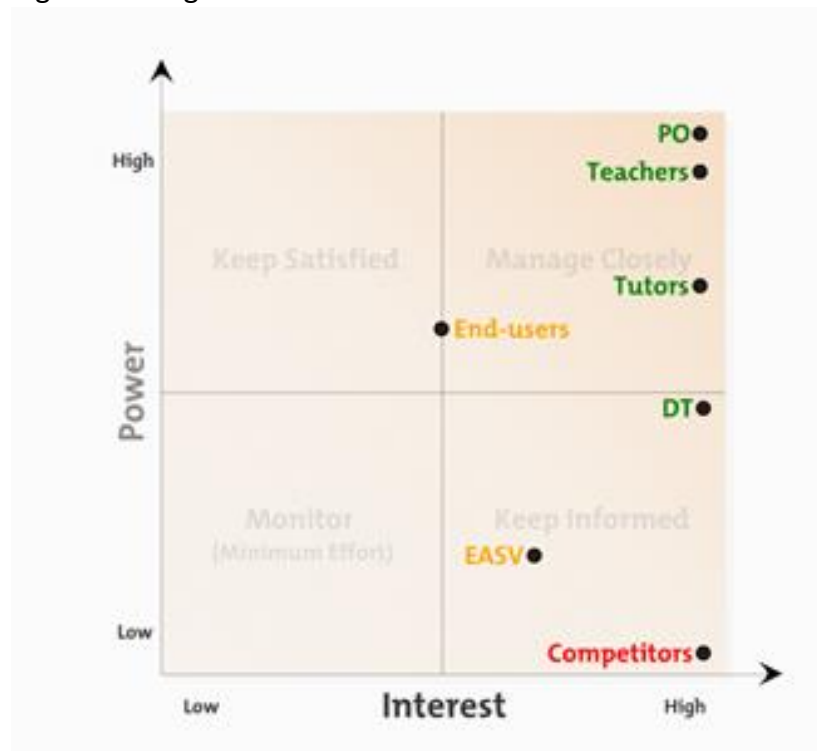


Figure 1. Power/Interest Matrix

### Risk analysis

Risk management is part of everyday business operations and project implementations. Despite the uncertain and unpredictable environment, businesses tend to start new, ambitious projects, therefore it is essential to map out the potential risk factors. Important to note that, it is most likely that not all of the risks could have been assessed and managed during a risk analysis, but it helps get prepared and eliminate the most certain ones.

#### Risk analysis

The first step of the risk analysis is to identify the potential threats and estimate the risk. The team decided to use one of the tools that have been provided during the ITO classes. In order to analyse and assess these threats, the group used the 'Risk Impact/Probability Chart' as a



starting point (see Appendix B) and listed them using 3 colours to visually highlight them better. The list of risks was objectively rated together based on previous project experiences.

### Risk management

Identified Risk	Risk Management
<b>Human-related:</b>	
Team sickness/injuries	This risk is a very unpredictable factor, but the team must count on this, and be prepared particularly because in the COVID-19 pandemic it can easily happen that someone became ill. In the case of long-term sickness, the DT must accept it and redelegate the tasks as fast as it is possible to control the risk. The protocol is the same if an injury happens and someone is slowed down and need “extra hands” from the time to avoid deadline delay. Since Github is used for the program’s implementation process, all data is shared and visible to the team members.
Team communication issues	Communication is a key in group works and effective communication allows the member to see through the complex working process. Several issues can occur in internal communication such as disagreements, conflicts, noises, uncertainty, etc. To minimize/ avoid them, the team decided to have online daily meetings because previous experiences shed light on that meeting “in person” is more effective than writing in the common chat group. Since it is not a new group and members executed other projects together before, huge conflicts will be less likely to occur. Members continue to stick to their well-established communication plan, which is part of their working agreement too.
External factors	Unexpected external factors (travel, moving, jobs, etc.) in private life are also a possibility to happen and the team must be informed as soon as possible about them. It must be noted that the DT members are dedicated to this project as their current priority. But external factors are highly unpredictable, so the best strategy for the team is to accept them and control them by reorganising the workload or find other solutions together.
<b>Project-related:</b>	
Code stuck	Code stuck is quite often happening in programming and could be easily eliminated by asking the team members or seeking help from the teachers/tutors to avoid time waste.
Avoiding/asking help too late	This is related to the former mentioned (code stuck) and has a much higher risk factor. Detective actions must be taken to avoid procrastination that could lead to deadline delay and affect the outcome negatively. Therefore, the team internally tracks the estimated timeframes and in case of detected deviation help/ guidance will be involved immediately.
Deadline delay	One of the highest risks because the team has limited time constraints for the project and any kind of deadline delay could permanently jeopardize the project’s result. By taking again detective action and modify SCRUM by taking a look at the Burndown chart would prevent the team from deadline delay. For example, someone can overtake the task and the others can move further with other tasks.

	Since deadline extension is not an option, the most important task is to avoid delays. It is not possible to achieve everything that is planned because of the deadline, so the DT has accepted this too and seeks to take under control the prioritizes and choose only the most important requirements for the program.
Quality issues	Feeling uncertain regarding the quality of the implemented codes is another thing that is quite common in programming. However, with using the 3 layered architecture and DoD this risk should be avoided because the adequate quality is ensured.
<b>Tech-related:</b>	
Technical difficulties	This risk is controlled due to all the collected data is available online, thus difficulties or destruction of electronics will not affect the project except there is no internet connection/ problem occurs with the server.
<b>Reputation-related:</b>	
Loss of client	The absolute highest threat is losing the customer which could be controlled by focusing on two elements. Firstly, loss of client could happen because of the high competition with other groups. The only way to resolve this is to compile the best possible product. Secondly, if the customers will not like the program because it is not matching with their expectations the team will lose them. Due to the limited communications with the clients (2 x 20 min. meetings) it is hardly possible to get in-depth feedback on the program. That's why the DT must proactively listen to the given feedback and find agile ways to implement the best solutions for fulfilling the client's needs.

## 2. Pre-game

### 2.1. Project organization

We started the project by creating a working agreement (Appendix C). We wanted to make sure that each of us had similar ambitions and was going to put the same amount of work. Also, we decided to meet every day (at least on the group chat) to be aware of everyone's status and help each other if needed. Then, after carefully reading the requirements, we discussed how we believed the program should look like. At last, we created the initial product backlog. There, we pointed out general biggest points that would make the base for the future development.

### 2.2. Overall project schedule

Date	Time	Activity
Monday 26/4	9:00	Kick-off meeting for both class A and B.  9:00 - Kick off on Zoom
Friday 7/5		1 <sup>st</sup> SCRUM review meeting. 20 min per group.
Thursday 27/5		2 <sup>nd</sup> SCRUM review meeting. 20 min per group.
Tuesday 1/6	14:00	<b>Hand-in the report and program on WiseFlow</b>

## 2.3. Initial product backlog

As an admin I want to create new screens	31 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to be able to switch between the views	17 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to manage users (add, delete, edit)	13 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to be able to edit the screen (template)	9 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to log in	9 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As and admin I want to be able to see a preview of the active screen (template)	8 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to delete the screens	2 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As an admin I want to assign users to screens	8 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As a user I want to log in	2 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>
As a user I want to see the screen	5 hours	Sprint completed	<div><div></div><div></div><div></div><div></div><div></div></div>	100%	<div><div></div><div></div><div></div><div></div><div></div></div>

## 2.4. Architecture

We applied 3-layer architecture and MVC pattern. Also, we did our best to apply SOLID principles and general good practices for writing robust software.

Three layer architecture consists of Graphical User Interface, Business Logic Layer and Data Access layer. For each layer group created another package. In order to decouple both classes and layers we decided to use Facade pattern combined with interfaces. This solution introduced additional complexity. Moreover, maintenance of the system became more time consuming.

Graphical User Interface contains MVC pattern. Whenever any action in the view is invoked, controller is informed and then request is sent to the model and then goes all the way down. In 3-layer architecture communication goes only in one way. From the highest layer to the lower ones.

Business Logic Layer is responsible for all the logic in the application. All the searching, sorting, checking conditions is placed in this layer.

Data Access Layer is responsible for manipulating data that is stored in database and locally in the files system. In this package we do CRUD operations in database, we convert pdf to html, and we save and delete files.

One of the challenges was to implement a system that will be synchronized on many devices. The problem lied in the way in which we were going to inform other instances of the program whenever any change happens. Our first thought was to send information to other devices that change occurred and only in that situation. We managed to do that but then there was another question how other instances would know that change occurred.

The next observation was that no matter how many different instances of the program we have, they all are connected to the one external database. Then all the information that system has to be updated comes from database.

In our architecture we implemented Observer, Command, Singleton, Memento, Object pool and Facade pattern.

It's important to mention even now that Singleton has numerous drawbacks. In this project we decided to use it anyway due to simplicity of using it.

## 3. Sprint 1

### 3.1. Sprint planning

During our first group meeting we decided to focus only on the admin side. We were going to start from implementing first functionalities in the *Create new, Users* and *Templates View*. We agreed that each of us would create a prototype and then we would brainstorm, compare our ideas and create a clear vision for our program design. After doing so, we went through an extensive research to make sure that our plans would be possible to realize with our current skills.

Also, we agreed that at that point, for our state of understanding, we were not able to plan the database model. We agreed to go back to that task after we get a better understanding of the problem we were trying to solve.

#### *User stories for sprint 1*

Admin	As an admin I want to be able to switch between the views	17 h	Sprint completed
Admin	As an admin I want to create new screens	31 h	Sprint completed
Admin	As an admin I want to manage users (add, delete, edit)	13 h	Sprint completed
Admin	As an admin I want to be able to edit the screen (template)	9 h	Sprint completed
Admin	As an admin I want to log in	9 h	Sprint completed
Admin	As an admin I want to delete the screens	2 h	Sprint completed
Admin	As and admin I want to be able to see a preview of the active screen (template)	8 h	Sprint completed

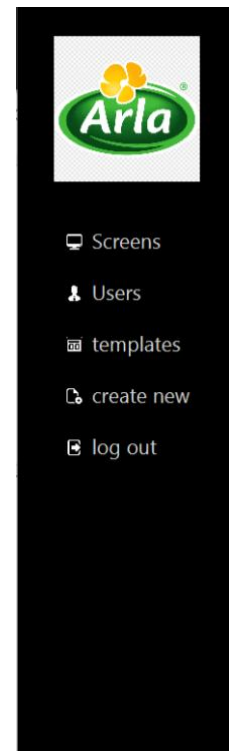
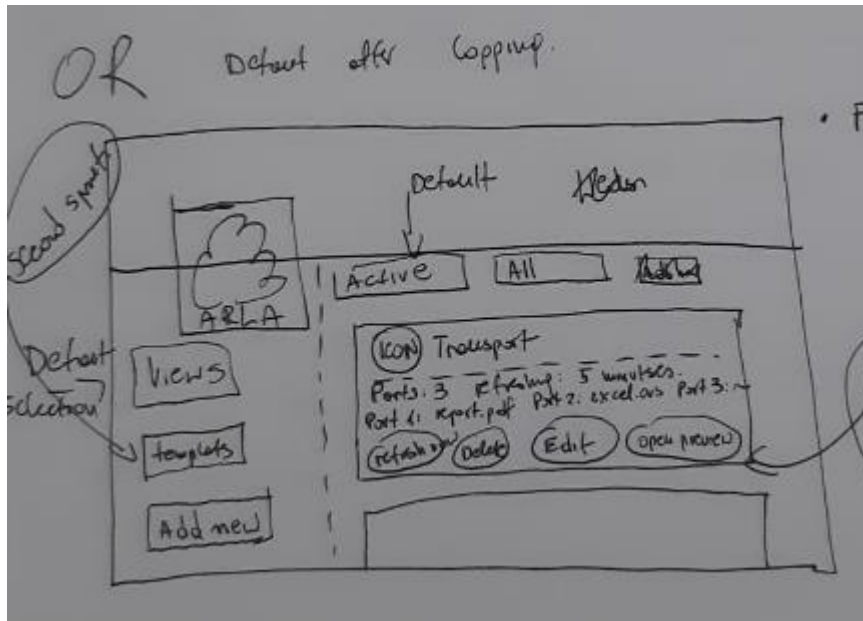
### 3.2. GUI (including UI-design patterns)

As mentioned above, we decided that each of us would study the program on their own and create a prototype for the graphical user interface. It helped us look at the project from 3 different perspectives and take everyone's ideas into consideration.

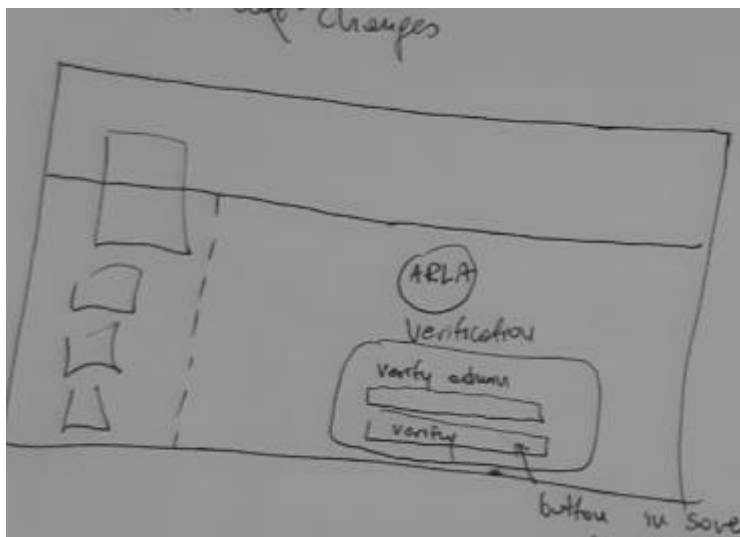
After discussing each prototype, we agreed to implement the following:

1. The main window with the navigation column on the left. It contains buttons that are used to switch between windows: in *Views* we were going to show information about the screens; in *Templates* we were going to add few layouts and enable admin to create from them; in *Add new* admin would be able to create totally new screen. We

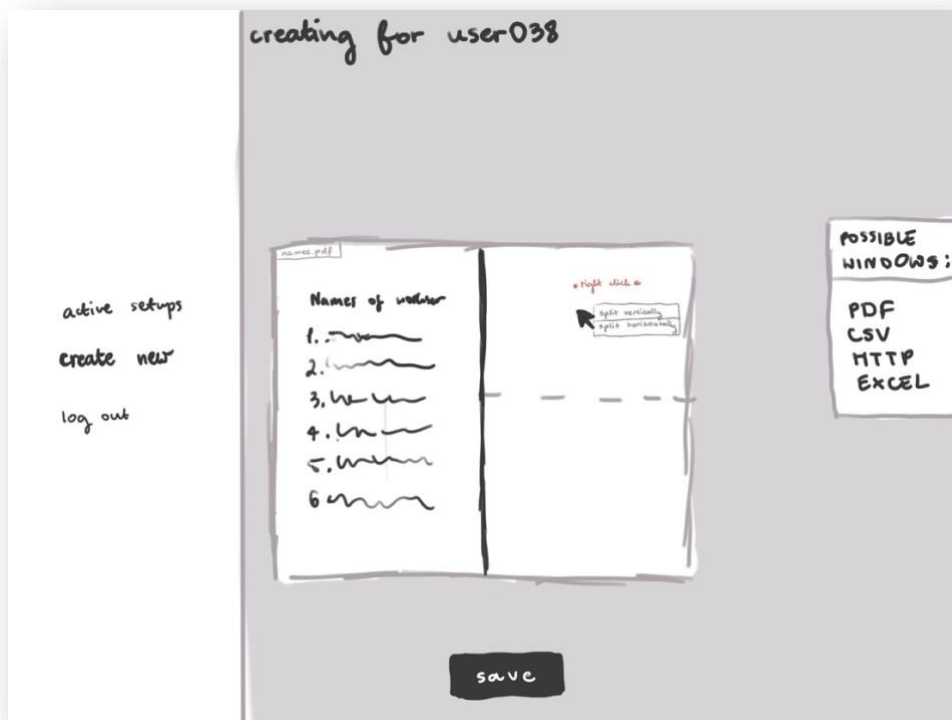
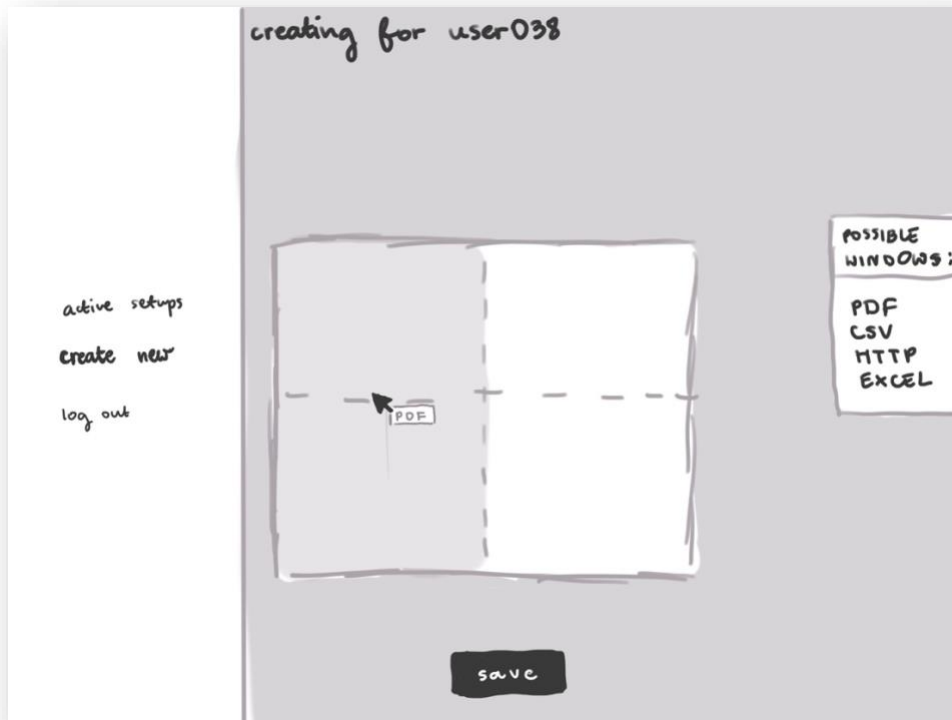
also decided to add *Users* and *Log out* labels. (Later we gave up on having *Templates* and changed the names from *Views* to *Screens* and from *Add new* to *Create new*).



2. Log in functionality inside the window. That screen was going to appear when user starts the program and if any important change is about to be made.

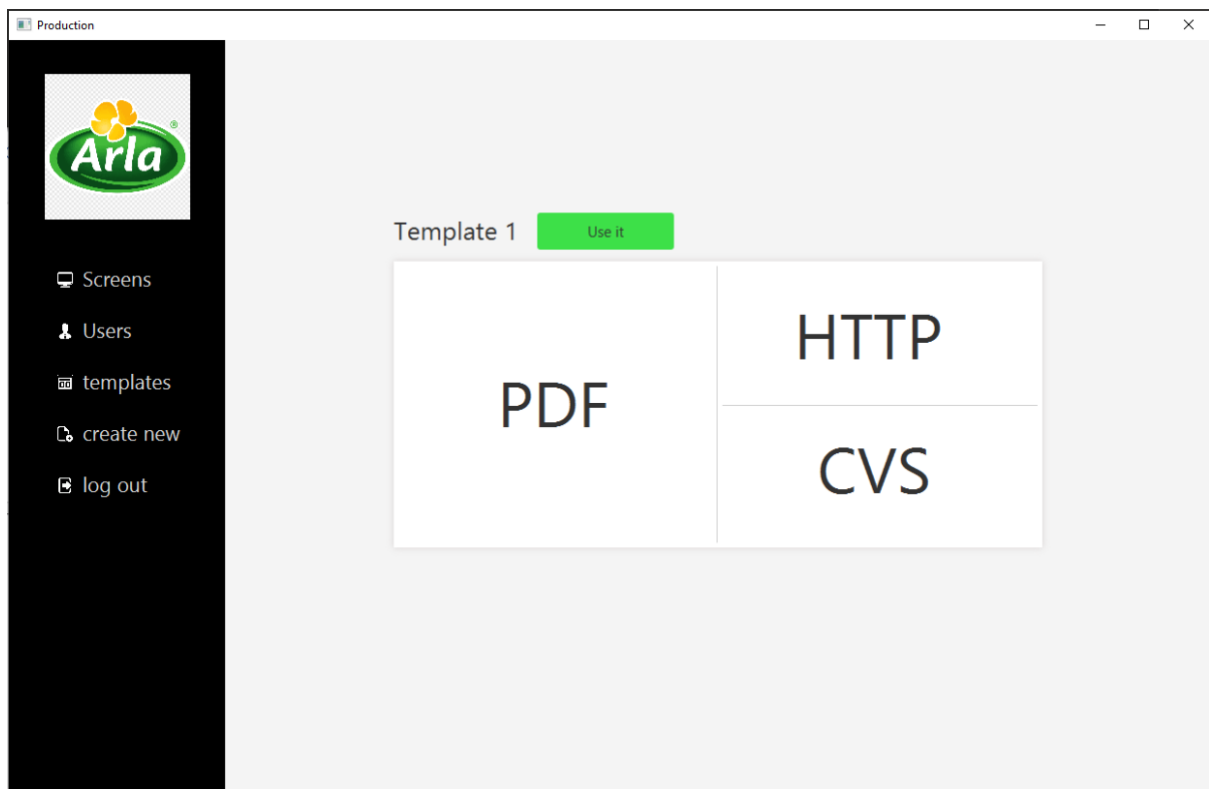


3. Creating new screen would start with a 'base' (GridPane with 2 columns and 2 rows). Admin can drag the file type label from the menu on right side onto the box they want to fill in. Then, they can upload the specific file. If they want to split the box, they can do so by choosing from context menu activated by right mouse click.

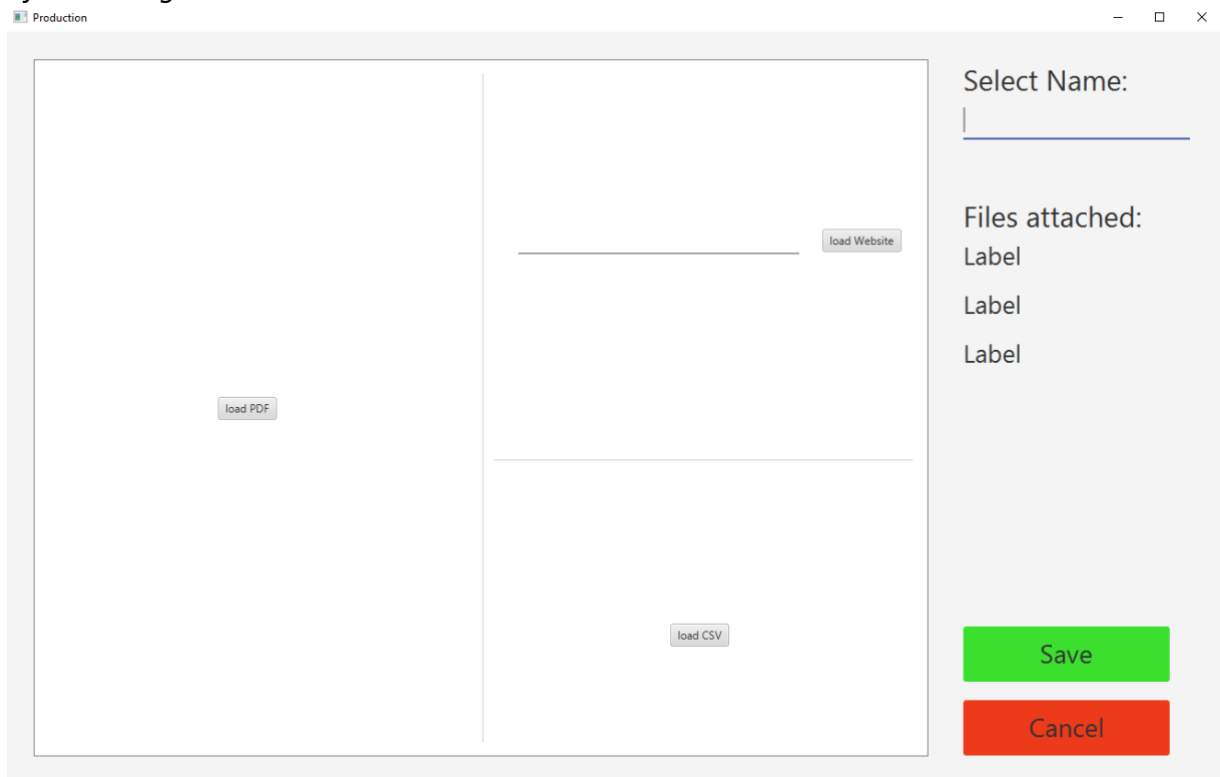




#### 4. Templates



*After clicking "use it" button*



## 5. Users

Arla

Arla Admin View

Q

Client 1
Client 2
Client 3
Client 4
Client 5
Client 6
Client 7
Client 8
Client 9

Details

Name: \_\_\_\_\_

Telefon: \_\_\_\_\_

Email: \_\_\_\_\_

Address: \_\_\_\_\_


Add

Delete

Edit

Log out

## 6. Screens



name

Refreshing: 5 mins

Attachment 1: pict.png

Attachment 3: pict.png

Attachment 2: howTo.com

Attachment 4: pict.png

Refresh now

Delete

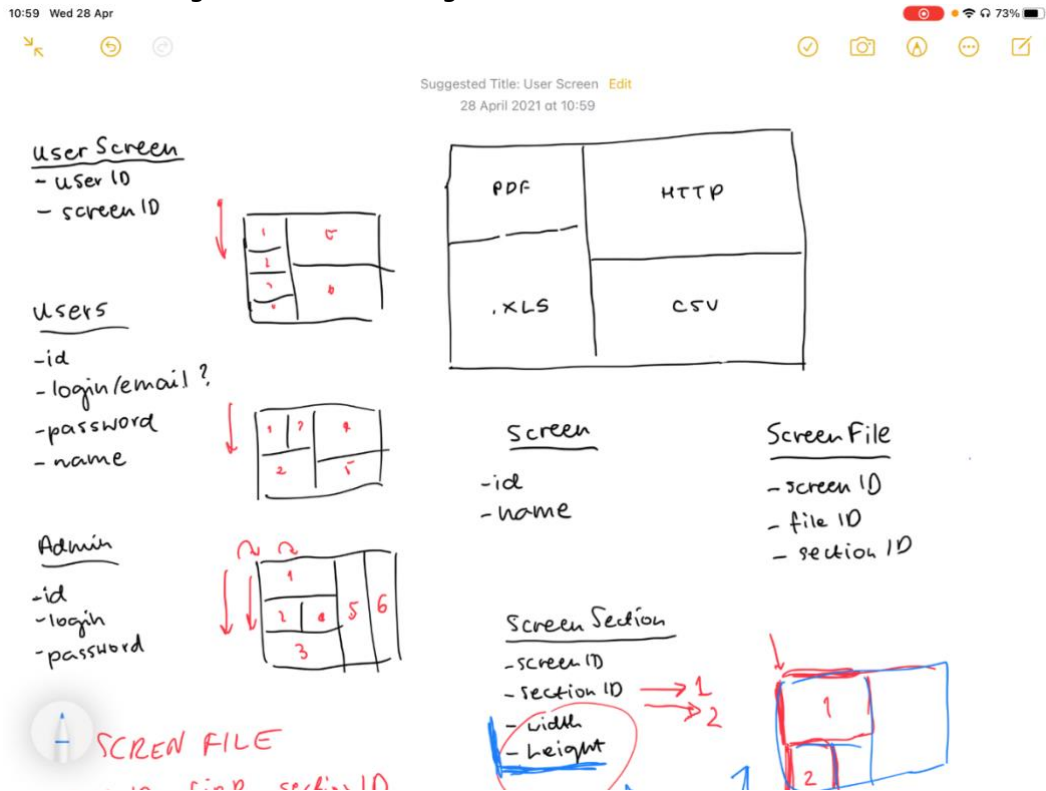
Edit

Open Preview

### 3.3. Data model

After we spent some time trying to understand what we needed to do, we attempted to make our first database design. Later, it turned out to have many flaws and we ended up changing it in following sprints. The cause for that was that at that point we didn't pay enough attention to GridPane documentation, which we planned on using. Also, we didn't predict to come across some problems with saving the files for each section.

#### Brainstorming the database design



#### Initial business entities design:

##### UserScreen

- userID
- screenID

##### Admin

- id
- login
- password

##### User

- id
- login
- password

##### Screen

- id
- name

##### ScreenSection

- screenId
- sectionId

- width
- height

##### ScreenFile

- screenId
- sectionId
- fileId

##### FilePath

- id
- path

### 3.4. Implementation

#### DefaultScreen

This view was provided as a template. Screen was divided into three different sections, each for: CSV, PDF and website. There wasn't any challenge of creating this layout so we could fully focus on functional parts as loading pdf and csv files.

From the time perspective, implementing this template was very good move because we could focus on functional part and database design would evaluate anyways, consequently when conquering new challenges.

Firstly, we created a table called *DefaultScreen*. It was the only table that wasn't changed and eventually it was discarded due to many drawbacks of using screen based on that table.

DefaultTemplates			
	Column Name	Data Type	Allow Nulls
▶	id	int	<input type="checkbox"/>
	name	varchar(55)	<input type="checkbox"/>
	destinationPathCSV	varchar(255)	<input type="checkbox"/>
	destinationPathPDF	varchar(255)	<input type="checkbox"/>
	insertedWebsite	varchar(255)	<input type="checkbox"/>
			<input type="checkbox"/>

Then, one team member could add functionalities for loading another types of files and simultaneously another team member could implement functionality for creating a screen with more control, where we can merge cells and create custom views.

Having created *Create new* functionality we got completely other perspective on the database design. We rethought that and created two tables: Screens and ScreenSections. Rows could be easily mapped in program because GridPane uses indices and span constraints.

#### Base for the create new screen functionality

We managed to provide merging functionality at the very early stage of the process. In next sprints we used this same methodology, but unfortunately if statements couldn't be kept so simple.

By default, we were defining that menu items for merging should be disabled. Then in the first if statement(line 222) we were checking if clicked element was in the row zero. In second if statement we were checking if clicked element was in column zero. We could do that because at that point we used GridPane with prefixed number of columns and rows to two.

In on actions we simply increment either row span or column span.

```

216 + private void showContextMenu(MouseEvent mouseEvent, Node node) {
217 +     if (mouseEvent.isSecondaryButtonDown()) {
218 +
219 +         right.setDisable(false);
220 +         down.setDisable(false);
221 +
222 +         if(GridPane.getRowIndex(node)!=null) {
223 +             if (GridPane.getRowIndex(node) == gridPane.getRowCount() - 1) down.setDisable(true);
224 +         }
225 +         if(GridPane.getColumnIndex(node)!=null) {
226 +             if (GridPane.getColumnIndex(node) == gridPane.getColumnCount() - 1) right.setDisable(true);
227 +         }
228 +         contextMenu.show(node, mouseEvent.getScreenX(), mouseEvent.getScreenY());
229 +
230 +         right.setOnAction((event) -> {
231 +             int span = 0;
232 +             if (GridPane.getColumnSpan(node) == null) span = 1;
233 +             GridPane.setColumnSpan(node, span + 1);
234 +             node.setStyle("-fx-background-color: GREEN"); //to check if worked
235 +         });
236 +
237 +         down.setOnAction((event) -> {
238 +             int span = 0;
239 +             if (GridPane.getRowSpan(node) == null) span = 1;
240 +             GridPane.setRowSpan(node, span + 1);
241 +             node.setStyle("-fx-background-color: BLUE"); //to check if worked
242 +         });
243 +     }
244 + }

```

## PDF

We spent a significant amount of time finding the best open-source library that could be used to open PDF files. We didn't find even one that could be used both gracefully and would be simple in the implementation. Then we changed direction and instead of using libraries we decided to do something else - we save PDF file and then convert it to HTML file. HTML file can be easily opened using WebView.

## Drag and drop functionality

We believed implementing this functionality would be very beneficial for the user experience. They can easily move the file type label to chosen section in the screen (as shown before on the prototype in chapter 3.2. GUI). We are using DragEvent. (More in code examples).

At the end we open one of the loading methods (depending on the dragged label with file type name). There we open File Chooser and load the specific file using one of our utility classes.

## CRUD for managing users:

The user view contains a Combobox, a search bar, 3 buttons (delete, add, edit), a tableview, and two hidden panels.

### Combobox:

The Combobox contains 3 options (All, admins, users). By selecting these, we can change the content published in the tableview. If the admin only wants to see the list of users or the list of admins or both of them, there is an option to do that.

```
public void comboBoxSelect(ActionEvent actionEvent) {  
    JFXComboBox comboBox = (JFXComboBox) actionEvent.getSource();  
    String selectedItem = (String) comboBox.getSelectionModel().getSelectedItem();  
    ObservableList<User> selectedType = returnSelectedUsers(selectedItem);  
    userTableView.setItems(selectedType);  
}
```

### Search bar:

With the search bar, it is possible to specifically search for users and admins in the tableview. As the admin starts typing the name of the user/admin that should be found, the tableview content would change according to that simultaneously.

```
public void search(){  
    FilteredList<User> filteredData = new FilteredList<>(FXCollections.observableList(userModel.getAllUser()));  
    userTableView.setItems(filteredData);  
    searchField.textProperty().addListener((observableValue, oldValue, newValue) ->  
        filteredData.setPredicate(userModel.createSearch(newValue))  
    );  
}
```

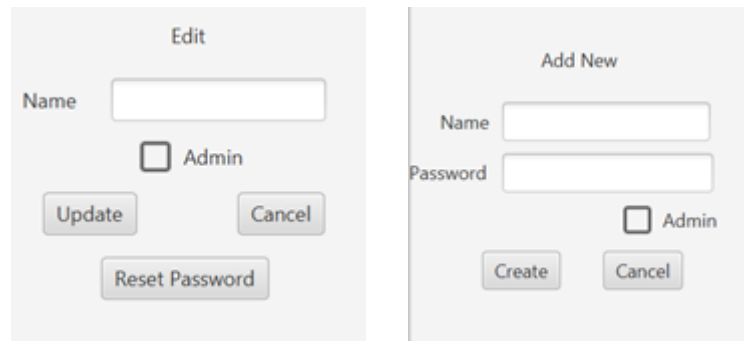
### The 3 buttons:

There are 3 different buttons, and each performs a different task.

1)Delete button: it deletes the chosen user/admin from the database. This has been given an extra function at the request of clients, this is the confirmation window. Confirmation is required for each deletion.

```
public void btnDeleteUser(ActionEvent actionEvent) {  
    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);  
    alert.setTitle("Confirmation Dialog");  
    alert.setHeaderText("Are you sure about deleting this user?");  
    |  
    Optional<ButtonType> result = alert.showAndWait();  
    if (result.get() == ButtonType.YES){  
        // ... user chose OK  
        User selectedUser = userTableView.getSelectionModel().getSelectedItem();  
        userModel.delete(selectedUser);  
    } else {  
        alert.close();  
    }  
    userModel.loadUsers();  
}
```

2) Edit button: is used to display the edit anchor pane and it is the same for the 3)Add button too.



There is a cancel button on the edit and on the add new user anchorpane, with the help of this, we can hide the two anchorpane again. In the case of edit, it also includes an update, a reset password function, and an admin check box. These are for the admins to be able to implement the user/admin names or to permit access to the users to be admins and vice versa. A reset password is an option for if anything happens to a user's password (forgetting it or something goes wrong), the admin can reset it. When the next time this user wants to log in and enters the password into the password field, what has been entered will be the new password. The admin can change the name of the users/ admins by uploading the name of the previously selected admin/user into the Name field, and the new name entered in its place will be uploaded to the database.

The operation of the update button:

```
public void setCreateAdmin(boolean isAdmin) {
    User newUser = userTableView.getSelectionModel().getSelectedItem();
    newUser.setUserName(editNameField.getText());
    newUser.setAdmin(isAdmin);

    userModel.updateUser(userTableView.getSelectionModel().getSelectedItem(), newUser);
    userModel.loadUsers();
}

public void btnUpdate(ActionEvent actionEvent) {
    boolean isAdmin = true;
    if (editAdmin.isSelected() == false)
        isAdmin = false;
    setCreateAdmin(isAdmin);
}
```

The method of displaying the edit anchorpane and it is the same for the add user too.

```
public void btnEditUser(ActionEvent actionEvent) {
    TranslateTransition show= new TranslateTransition();
    show.setDuration(Duration.seconds(0.4));
    show.setNode(editTable);
    show.setToX(0);
    show.setToY(100);
    show.play();

    editTable.setTranslateX(0);
    editTable.setVisible(true);
    addNewUser.setVisible(false);
    edit.setDisable(true);
    add.setDisable(false);

    add.setOnMouseClicked(event ->{
        show.setNode(editTable);
        show.setToX(0);
        show.setToY(-100);
        show.play();

        addNewUser.setTranslateX(0);
    });
}
```

The add anchorpane got a create and a cancel button, which works the same way as the edit button. Also, it got an admin checkbox, which is used when someone wants to create a new user/admin, so the text written in the Name field will be the username, and the text written in the password field will be the password. If we select the admin checkbox, we will create a new admin. If we don't do that, a user will be created.

### 3.4.1. Code examples

*Drag and drop for the file type labels*

```
private void setOnDrags() {
    jpgL.setOnDragDetected(event -> dragStart(event, jpgL));
    pngL.setOnDragDetected(event -> dragStart(event, pngL));
    pdfL.setOnDragDetected(event -> dragStart(event, pdfL));
    csvL.setOnDragDetected(event -> dragStart(event, csvL));
    httpL.setOnDragDetected(event -> dragStart(event, httpL));
    xlsL.setOnDragDetected(event -> dragStart(event, xlsL));
    videoL.setOnDragDetected(event -> dragStart(event, videoL));
}
```



```
private void dragStart(MouseEvent event, Label source) {  
    Dragboard db = source.startDragAndDrop(TransferMode.ANY);  
    db.setDragView(source.snapshot( snapshotParameters: null, writableImage: null));  
    ClipboardContent cb = new ClipboardContent();  
    cb.putString(source.getText());  
    db.setContent(cb);  
    event.consume();  
}
```

```
private void dragOver(DragEvent dragEvent) {  
    if (dragEvent.getDragboard().hasString()) {  
        dragEvent.acceptTransferModes(TransferMode.ANY);  
    }  
}
```

```
private void dragDropped(DragEvent event) {  
    Dragboard db = event.getDragboard();  
    Node node1 = event.getPickResult().getIntersectedNode();  
    Node node = getNodeToUse(node1);  
    if (db.hasString()) {  
        switch (db.getString()) {  
            case "HTTP" -> loadHTTP(node);  
            case "PNG" -> loadImage(node, event);  
            case "JPG" -> loadImage(node, event);  
            case "PDF" -> loadPDF(node);  
            case "CSV" -> loadCSV(node);  
            case "XLSX" -> loadExcel(node);  
            case "VIDEO" -> loadVideo(node);  
        }  
    }  
}
```

*Getting screenId for Sections from saved screen*

```
//set proper id for that screen
try(ResultSet generatedKey = preparedStat1.getGeneratedKeys()) {
    if(generatedKey.next())
        screenID = generatedKey.getInt( columnIndex: 1);
    else
        throw new DALException("Couldn't get generated key");
}
```

### 3.4.2. Design Patterns/principles

#### Introduction

During the first sprint we implemented three patterns: singleton, command and facade. In this paragraph we will cover only the first two.

#### Problem Statement

We had to implement loading different screens corresponding to what user wants to open. In addition, sometimes there is a need to ask admin to provide credentials.

#### Open/Closed Principle

Thanks to using that pattern the load screen part is opened for extension but closed for modification. We can easily add new command if in the future we add new view and we don't need to circle back to one massive class in which a lot of code is duplicated.

#### Undo/Redo

We take advantage of that. In the initial planning, we thought it will be needed to check admins identity many times, whenever they make important change. Then there was a problem that we can load Log in screen but then we need to know to which view we need to go back. We solved that problem by combining memento pattern with command pattern.

Later we changed the vision and decided to ask a user for credentials only once, when logging in. However, we left implemented solution because it is completely functional.

#### Single responsibility principle

Although we use inheritance we can say that we follow this principle. Each class is responsible for only one action and if needed it overrides method of super class.

#### Implementation

We use one abstract class called Command which contains fields and one method for doing rollback. It uses Command manager which will be mentioned later.

```
public void rollback(BorderPane borderPane){  
    CommandManager.getInstance().getPrevious().load(borderPane);  
}
```

### Singleton pattern

Using singleton pattern is very convenient. It's easy solution that enables us to make sure that we have only one instance of a class. Being aware of its drawbacks we decided to use it anyway due to simplicity.

### Drawbacks

One of them is giving global access to methods and fields (of course if they don't have other modifier). Another easy to conduct drawback is that we can have only one instance. (This same time it is a reason why we use that pattern). It can be problematic when making Unit tests

### Interface Segregation principle

We did our best to follow interface segregation principle. To enhance this principle even more we decided to make interfaces for Facade in BLL and DAL extend other small interfaces that are specific to some parts of program.

```
13 public interface IDALFacade extends IFile , IPDF, IScreen{
```

### Memento pattern

By definition, this pattern enables us to store and restore previous state of the program. In our case we store information which view was previously opened. Then when we know that we can easily do a rollback.

In the first sprint we used this pattern for logging functionality. Then we were planning to use it for validating user identity when making important changes. Also, it could be used to implement two arrows in the top left corner. One of them could be used to go back(undo) and another to go further(redo). Unfortunately, we didn't make those functionalities either in first sprint nor any other.

```

10 +public class CommandManager {
11 +    private Deque<Command> history = new LinkedList<>();
12 +    private static CommandManager instance;
13 +
14 +    public static CommandManager getInstance(){
15 +        if(instance==null)
16 +            instance = new CommandManager();
17 +        return instance;
18 +    }
19 +
20 +    private CommandManager() {
21 +    }
22 +
23 +    public void addCommand(Command command){
24 +        history.offer(command);
25 +    }
26 +
27 +    public Command getPrevious(){
28 +        return history.peek();
29 +    }
30 +}

```

### 3.5. Sprint Review

Our first sprint review was very beneficial. We received some valuable feedback from the client which set the course for us for the next sprints. The things we knew we had to change / improve were:

- Hiding the passwords in users view
- Giving the admin the possibility to assign users as admins
- Assigning screens to users

### 3.6. Sprint Retrospective

Things that went well:
We improved our understanding of the problem
Communication in a team and dividing the tasks.
Things to improve:
In first sprint we didn't finish a lot. We spent a lot of time discussing our ideas, prototyping and defining our vision. We had to speed up or we wouldn't finish product on time.
We both did underestimate and overestimate the time it will take us to finish some tasks. In the next sprint we had to think more carefully about estimating time for tasks.

## 4. Sprint 2

### 4.1. Sprint planning

After the first sprint, we noticed that we were not able to fully oversee in which direction the program development would go in. That is why we decided to break second sprint into two parts. They ended up being tightly coupled because in third sprint we were building on top of what had been created in second sprint.

While dividing the work we were able to easily separate tasks so that each person worked in a bit different part of a program at the same time. From our experience it works better. Team members could freely modify classes and conflicts on git appeared very rarely.

In this sprint we were going to work simultaneously on providing preview for the client and control over creating a new screen for the admin. Also, we wanted to give more options for opening CSV files.

#### *User stories for the 2<sup>nd</sup> sprint*

Admin		
As an admin I want to assign users to screens	8 h	Sprint completed
Client		
As a user I want to see the screen	5 h	Sprint completed
Client		
As a user I want to log in	2 h	Sprint completed
Admin		
As an admin I want to see all screens	7 h	Sprint completed
Admin		
As an admin I want to add more grids in create new	4 h	Sprint completed
Admin		
As an admin I want the users passwords to be protected	4 h	Sprint completed
Admin		
As an admin I want to assign other admins	5 h	Sprint completed
Admin		
as an admin I want to see more csv files	16 h	Sprint completed
Admin		
As an admin I want more control over created screen	25 h	Sprint completed

Before starting the 3<sup>rd</sup> sprint we went through the requirements once again because we felt like we had drifted away from the core of the program. We wrote down a list of mandatory things that still needed to be finished and we prioritized them in the 3<sup>rd</sup> sprint.

### *User stories for the 3<sup>rd</sup> sprint*

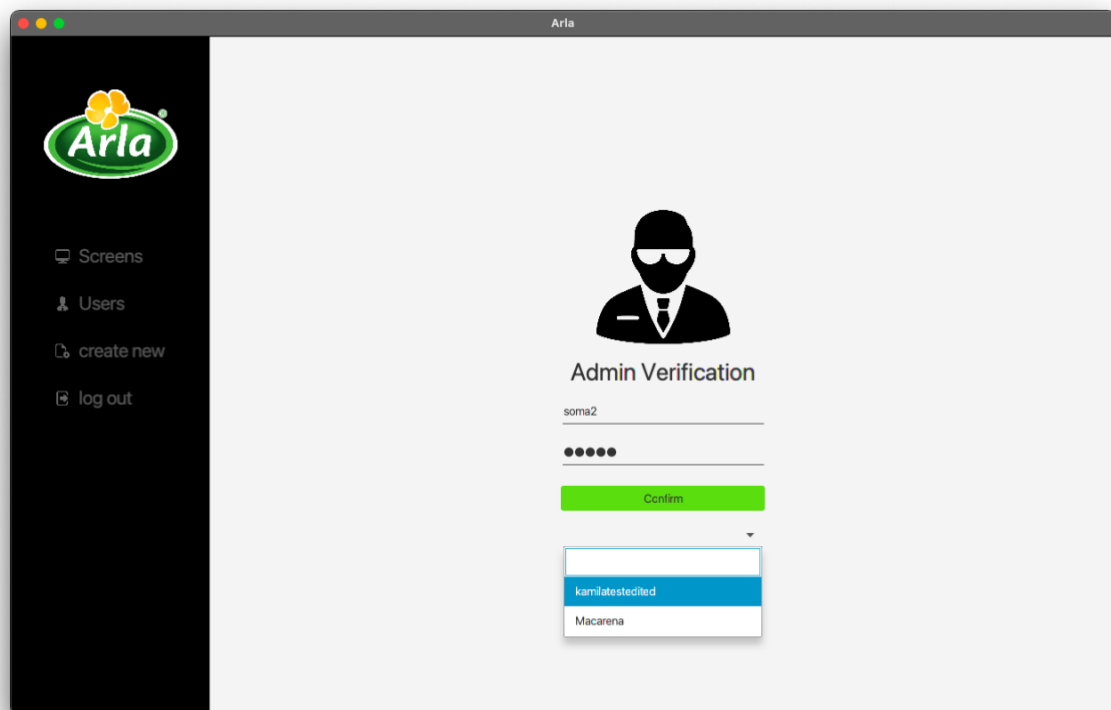
Admin	as an admin I want to add xlsx file	15 h	Sprint completed
Admin	as an admin I want to add video	12 h	Sprint completed
Admin	as an admin I want to assign many screens to one user	10 h	Sprint completed
Client	as a client I want to see updated files	6 h	Sprint completed
Client	as a client I want the grids to be zoomable	7 h	Sprint completed
Admin	as an admin I want to load the specific URI and keep it offline	6 h	Sprint completed
Admin	as an admin I need to protect the db connection / keeping everything updated	5 h	Sprint completed
Admin	as an admin I need to save the created screen	4 h	Sprint completed
Admin	As an admin I want to see updated all screens after editing	1 h	Sprint completed
Admin	as an admin I want to have better all screens view	5 h	Sprint completed
Client	as a client I need adjusted info about csv	5 h	Sprint completed
Admin	as an admin I want to see a preview for "puzzle" screen	8 h	Sprint completed

## 4.2. GUI (including UI-design patterns)

During 2<sup>nd</sup> and 3<sup>rd</sup> sprint we didn't make any significant changes to the GUI. We only improved and added some details as we were creating new functionalities. This included:

- Giving the user the freedom to set rows and columns at the beginning of creating new screen – also with the reset button to start from the beginning
- Context menu in *Create New* allows user to merge grids instead of splitting them
- Showing usernames of assigned users to each screen in *Screens* view
- Adding combobox at the login if user is not an admin and they have more than one assigned screen
- Deleting templates label from menu at the left (we didn't have enough time to finish that functionality)

*Log in*



Preview of a screen created by an admin and shown for the assigned user

The screenshot shows a web application window titled "kubaTest". It is divided into three main sections:

- Top Left:** A table with 6 columns: First name, Last name, age, gender, country, and subscribed. It contains 8 rows of data.
- Top Right:** A video player titled "bla.mp4" showing a video of a person's hand holding a small object.
- Bottom Left:** A bar chart titled "title" with "Index" on the x-axis (1 to 10) and a y-axis from 0 to 5. It shows three data series: "Mass (kg)" (red), "Spring 1 (m)" (orange), and "Spring 2 (m)" (green).
- Bottom Right:** A Google search results page for the query "website".

First name	Last name	age	gender	country	subscribed
Anne	Kolling	23.0	F	Croatia	yes
Tom	Dethor	54.0	M	Germany	yes
Dorothy	Ginde	21.0	F	France	yes
Gin	Deeths	34.0	F	France	no
Ben	Nordhy	37.0	M	Poland	yes
Rob	Molifng	25.0	M	Denmark	no
Sam	Werngt	42.0	M	Denmark	yes

View with all screens created by the admin

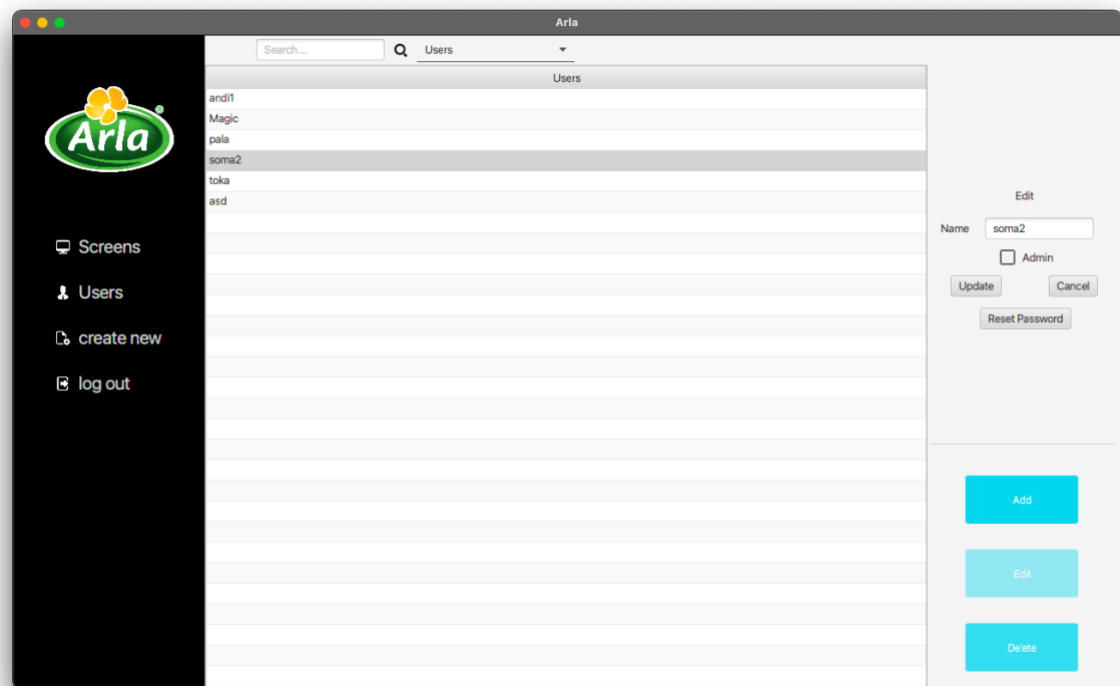
The screenshot shows the "Arla" admin interface. On the left is a sidebar with the Arla logo and navigation links: Screens, Users, create new, and log out. The main area displays a list of screens created by the admin, with tabs for "All" and "Used".

The list shows two screens:

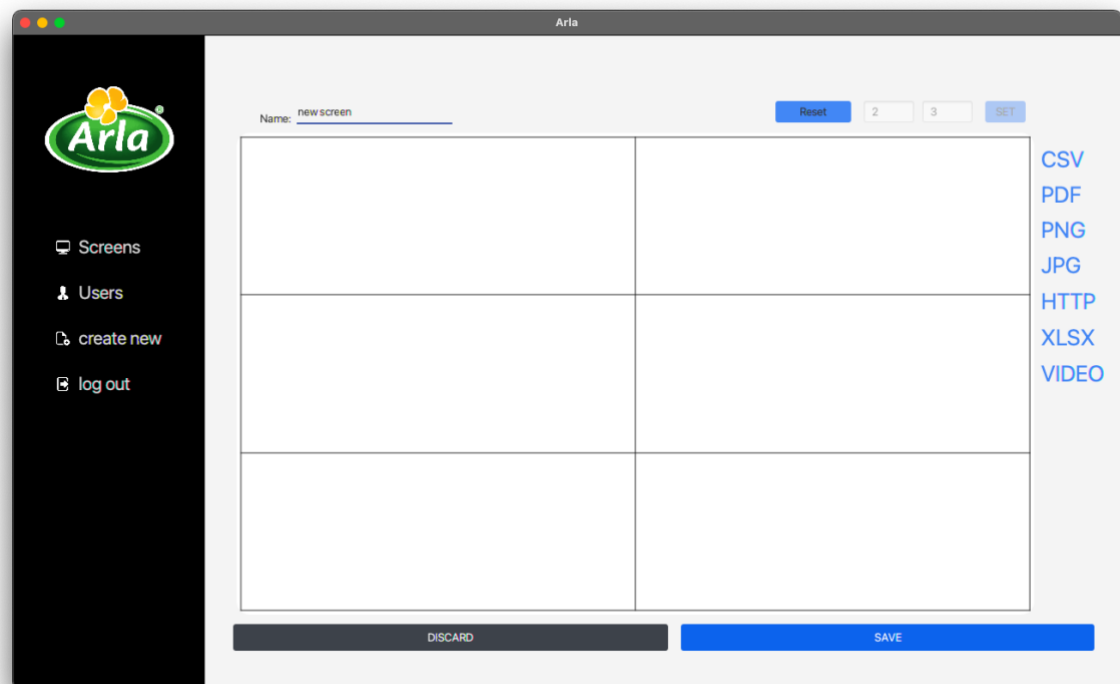
- kamilatestedited:** Assigned users: soma2. Refreshing: 5 mins. Attachments include CSVData/kilograms.csv, XLSDData/test.xlsx, cats, and PDFData/DB - Assignment 1.pdf. Actions: Refresh now, Delete, Edit, Open Preview.
- Macarena:** Assigned users: soma2. Refreshing: 5 mins. Attachments include ImageData/panda.JPG.jpg, ImageData/pict.png, CSVData/peopleMonths.csv, and XLSDData/test.xlsx. Actions: Refresh now, Delete, Edit, Open Preview.



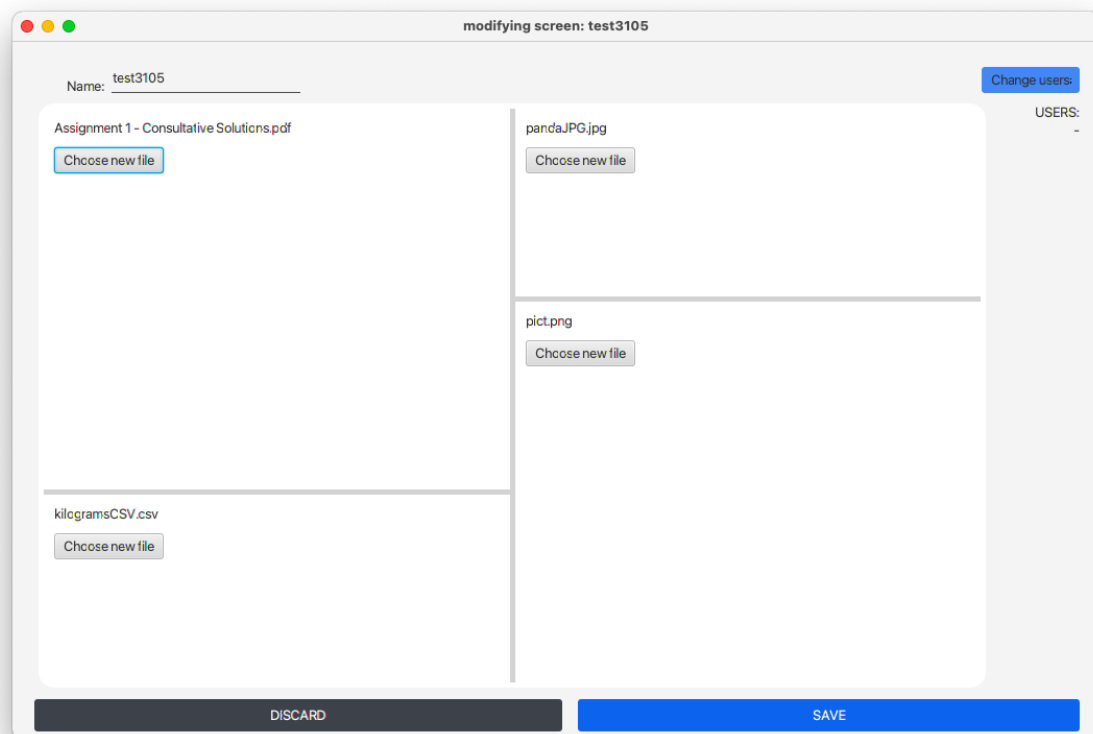
*Users view*



### Creating new screen



## Editing the screen



### 4.3. Data model

#### One-To-Many-Relationship (1:N)

Now we can say that we have one 1:N relationship (later we will see that actually there are three 1:N relationships). Screens is primary table and Sections is related table. Ideally in this relationship Screens can have 0, 1 or N sections. However in our program we force the user to set minimally one section. Every record in Sections can relate to only one record in Screens.

#### Many-To-Many-Relationship (N:N)

We have one N:N relationship. It is between Users in Screens. We don't know the direct way to establish direct Many-To-Many-Relationship so that actually it is divided into two 1:N relationships. As in the previous paragraph, Users and Screens are primary tables. Each of their records may be associated with 0 or more records in UsersAndScreens. However, each record in UsersAndScreens is always associated with only one record from Users and only one record from Screens.

#### Users

We realized that each user could be an admin. That's why we use the same table to store them. Field *isAdmin* lets us know if given user is an admin or not. *isReset* is used to reset the user's passwords. If this field is true then at the next log-in whatever will be typed into password field will be set as a new password. After that, the *isReset* value goes back to false. Apart from that we have id, username and password.

## UsersAndScreens

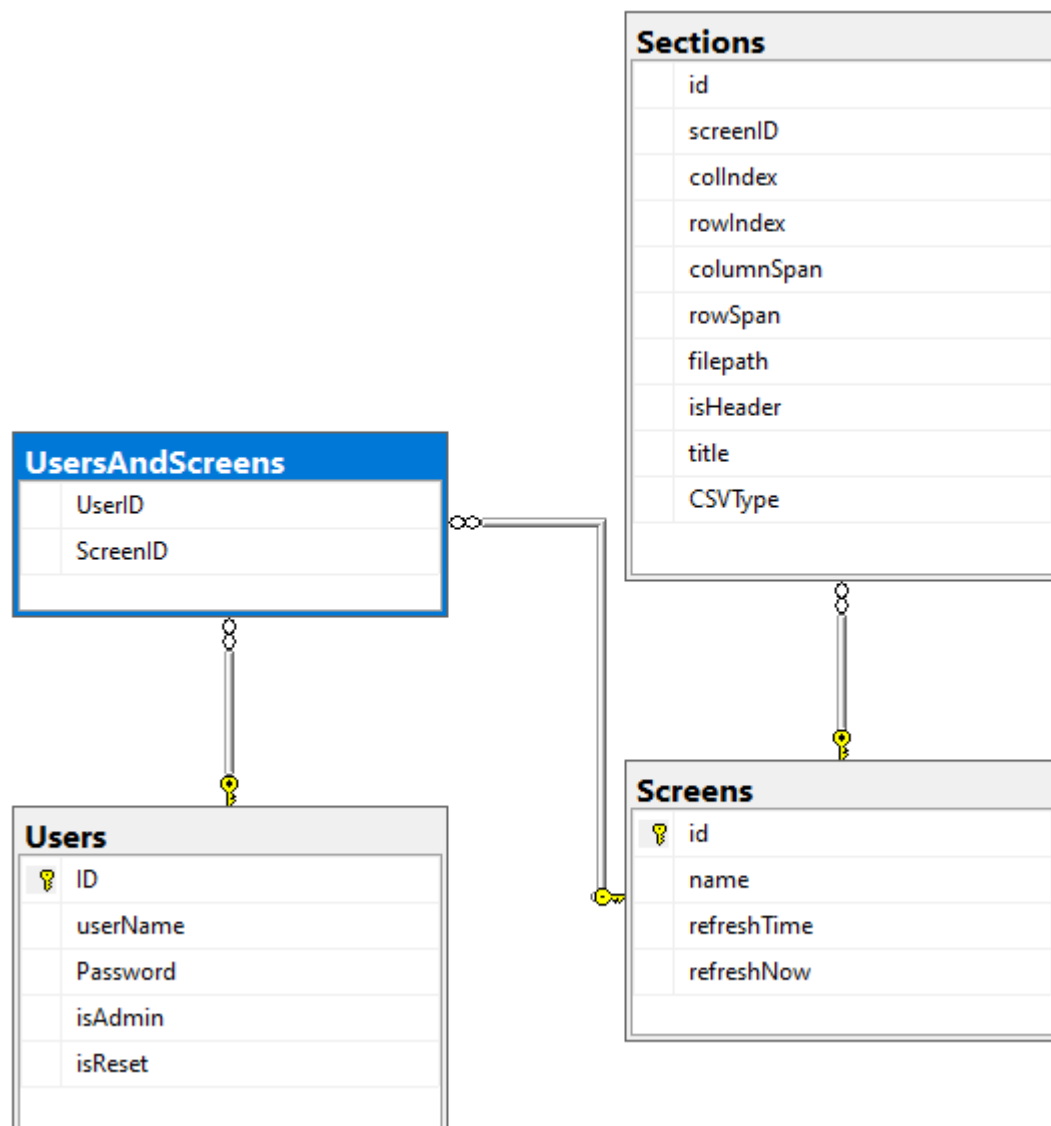
This table stores relations between users and screens created by admins. We needed that table cause Users and Screens have *many-to-many* relationship, which means that each screen can be assigned to many users and each user can be assigned to many screens.

## Screens

Screens have id, name, refreshTime and refreshNow. If the last one is true then all open specific screens will be forced to reload. Then value goes back to false.

## Sections

First, after their own id, we have screenId, which tells us which screen this section belongs to. Then, we have column and row Id, column and row span, which are the information easily accessible from GridPane. *FilePath* is the path for selected file which should be shown in that section. The last three – *isHeader*, *title* and *CSVType* – are rows used to describe how the csv file should be displayed. If given file isn't csv those three rows are null.



## 4.4. Implementation

### Create new screen

We improved creating screen functionality and gave a user much more control. At first, they define number of rows and columns in the top right corner. Then, they can freely merge cells without acquiring conflicts (if some cells are occupied, there shouldn't be any chance to fill them).

### Initializing after rows and columns are selected

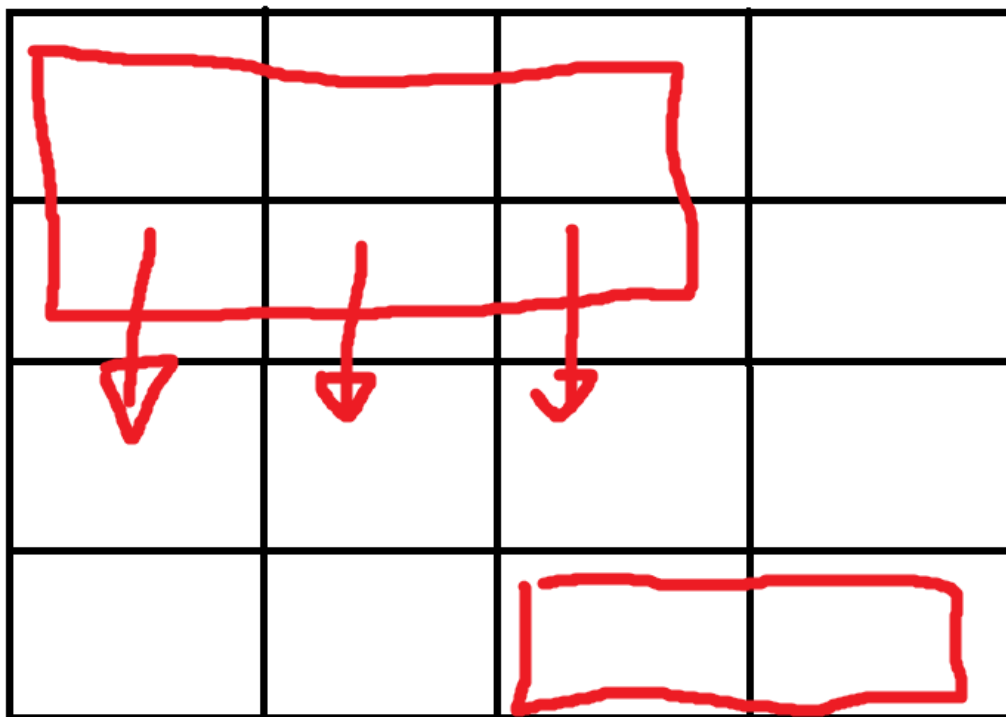
When button *set* is pressed, we initialize *gridPane*. We validate if inserted numbers are in correct range. If yes - we create cells and set constraints.

### Merging cells

Overall, we take one approach when merging down/right and another when merging left /up.

When we merge down we need to check few conditions. First one is if a node isn't in the last row (then checking other conditions wouldn't make sense). Then we check how many rows we have down to the end and we iterate through each of them. When iterating we check if the row is occupied all along column span of the node. On the illustration below we can see that the first row below the box isn't occupied. However, the second is occupied. It is only partially occupied but nevertheless we cannot expand the node there.

Then we simply increase the row span of the node and keep its indices in the same position.



When we go up checking condition goes in the similar way. However, expanding the size of a node is completely different. We delete the node on which event was invoked and then add a new node above. That node has increased column span.

We store information about the state of the screen being created in few ways. One of them is saving for each node information that create a box or is in a box information about it. For this reason we use class called "Information". This class had three fields. Later we will explain how we use them.

```
800 public class Information {
801     private Node node;
802     private boolean filled;
803     private String filepath; // or query. later url
```

In the implementation there is one important method called `getNodeToUse()`. This method was one of the mile stones. Before we explain the method it is important to introduce two keywords: "parent node", "child node". If a node wasn't merged, it doesn't have any relation. In other case it does. Parent node is a node that is used to fill the box and whose column span and row span are as big as a box. Children nodes are all the nodes that are a space of parent node but have other indices.

```
506 private Node getNodeToUse(Node node) {
507     if(getInformation(node).getNode()==null)
508         return node;
509     else
510         return getInformation(node).getNode();
511 }
```

In the method shown above, there is an if statement. If it is true and result of the left part of condition is null node isn't in any relation. If it isn't null it means that node is either parent or child. For the sake of simplicity, we don't have another condition checking if it's a parent or a child. Instead, if a node is a parent, we save for it "parent node" as well (node points to itself).

### Described in depth way to merge down

Firstly, when initializing gridpane cells we set on action for each cell.

```
196 node.setOnMousePressed(event → showContextMenu(event, node));
```

ShowContextMenu() method is responsible for checking four possible ways to merge (up, down, left, right). Below we can see method called from mentioned class that checks possibility to merge down.

As a first step we get node that will be used later. We can see in line 548 that we do that using already mentioned method. Then in line 549 we make sure that index of the parent node/ of a node isn't the last index. In line 551 we check how many rows there are below the selected box. It is important that we take into consideration the row span of the selected node. For

example, if we have gridpane with 5 rows and selected node has row span equal to 3, we should only provide possibility to merge two more rows.

Then we have for loop. In line 553 we get column span. GridPanes method for getting column span returns null if the column span is equal to one so that we decided to move that checking functionality to another method.

```
547 private void checkDown(Node node, MouseEvent event) {
548     Node parentNode = getNodeToUse(node);
549     if (GridPane.getRowIndex(parentNode) != gridPane.getRowCount() - 1) {
550         List<MenuItem> menuItems = new ArrayList<>();
551         int noToConnect = gridPane.getRowCount() - 1 - GridPane.getRowIndex(parentNode) - getRowSpan(parentNode) + 1;
552         for (int i = 1; i <= noToConnect; i++) {
553             int columnSpan = getColSpan(parentNode);
554             boolean check = checkDownIsOccupied(parentNode, i, columnSpan);
555             MenuItem menuItem = new MenuItem("connect with: " + i + " bottom");
556             setDownOnAction(parentNode, i, columnSpan, menuItem);
557             if (check)
558                 menuItems.add(menuItem);
559         }
560         contextMenu.getItems().addAll(menuItems);
561         contextMenu.show(parentNode, event.getScreenX(), event.getScreenY());
562     }
563 }
```

Then in line 554 there is a method `checkDownIsOccupied()`. We will present that method later, now we just need to mention that this method ensures that we can expand the box by “i” rows and we won’t conflict with another box. If we won’t conflict with another box we add that item to the list that will be showed.

In line 556 we invoke method showed below. We simply increase the span by the chosen “i”.

```
565 @ private void setDownOnAction(Node node, final int finalI, final int finalColumnSpan, MenuItem menuItem) {
566     menuItem.setOnAction(actionEvent -> {
567         int span = getRowSpan(node);
568         int columnIndex = GridPane.getColumnIndex(node);
569         GridPane.setRowSpan(node, integer: span + finalI);
570         setDownNumeration(node, finalI, span, finalColumnSpan, columnIndex);
571         node.setStyle("-fx-background-color: GREEN");
572     });
573 }
```

Then in the line 570 we see method `setDownNumeration()`. This method is shown below.

In the first for loop we iterate though all rows that this newly created box occupies (we go that that method after span is increased). In the next for loop we iterate though all columns this node occupies. For each of them **we set value both other from zero and this same for all elements**. After setting values we increment variable called *incrementedValue* to prepare for another merging. Using method `setParentNode()` in line 649, we set the parent node (the node that should be called when any action on this node will be invoked).

```

643     private void setDownNumeration(Node node, int finalI, int span,
644                                     int finalColumnSpan, int columnIndex) {
645         for (int j = GridPane.getRowIndex(node); j < GridPane.getRowIndex(node) +
646             span + finalI; j++) {
647             for (int k = columnIndex; k ≤ columnIndex + finalColumnSpan - 1; k++) {
648                 array[j][k] = incrementedValue;
649                 setParentNode(j, k, node);
650             }
651         }
652         incrementedValue++;
653     }

```

### Described a difference for merge left/ up

Below we can see a part of code for merging up. At first we would like to point out to line 522. In this line we completely delete previously created box. Then in line 521 we get a node which has a row in the desired position and column in the position of the node that we will delete in the next line. Then in lines 533 and 534 we set desired spans (column stays this same so we only increase span for rows).

```

519     menuItem.setOnAction(actionEvent → {
520         int goalRow = GridPane.getRowIndex(usedNode) - finalI;
521         Node useThisNow = getNodeByCoordinate(goalRow, GridPane.getColumnIndex(usedNode));
522         gridPane.getChildren().remove(usedNode);
523         for (int m = GridPane.getRowIndex(usedNode); m < GridPane.getRowIndex(usedNode) + getRowSpan(usedNode); m++)
524             for (int n = GridPane.getColumnIndex(usedNode); n < GridPane.getColumnIndex(usedNode) + getColSpan(usedNode); n++) {
525                 Node node1 = new AnchorPane();
526                 Information information = new Information();
527                 information.setNode(useThisNow);
528                 node1.setUserData(information);
529                 node1.setOnMousePressed(event1 → showContextMenu(event1, node1));
530                 //getInformation(node1).setNode(useThisNow);
531                 gridPane.add(node1, n, m);
532             }
533         GridPane.setColumnSpan(useThisNow, getColSpan(usedNode));
534         GridPane.setRowSpan(useThisNow, integer: getRowSpan(usedNode) + finalI); // it should work but who knows hahah
535         useThisNow.setStyle("-fx-background-color: orange");
536         setUpNumeration(useThisNow);
537     });
538     if (check)
539         menuItems.put(i, menuItem);

```

### Model, Threads and Observer pattern

We believe it was the hardest task to conquer conceptually. The problem was that if there is a file modified / new screen added / new screen deleted / refresh button clicked – changes had to appear in all opened admins and clients.

### Problem statement

As simple as it may be now it was the most different challenge we solved comparing to other projects. Besides the synchronization between different instances of the program there was another obstacle.

When implementing solution we wanted to follow MVC and 3-layer architecture and in some way data had to go up the layers. If there is a change, GUI has to be informed about it. It just cannot ask for it because it doesn't know what is a change and when it appears. In addition, even if we get to the GUI in some way we need to pass data from model to Controllers. It would also break architecture.

## **Solution**

Communication can go only the way down. Model needs to ask for data. Then it doesn't know when to ask so it just asks all the time in some time intervals. Then model sends request to Business Logic Layer to analyze if there are any items added / deleted / any CSV file was changed.

Then we get three lists. List of added screens, list of deleted screens and list of screens that contain modified CSV file.

For information exchange between controllers and model we use Observer pattern. We will cover that section in paragraph called Design Patterns.

## **Model**

In Model-View-Controller pattern the model is used to store data and information about the state of application. Then we decided that this class will be Observable. This same time model is the only class that can be observable and won't break MVC pattern.

## **Listening for changes**

Here is the first place where we had to use our knowledge about concurrency. During all the time when user(both admin and client) is logged in, we are each few seconds sending query to database and then processing the data. If this process wasn't held in another thread, all GUI would be completely unresponsive during all the runtime of the program (it's never ending process).

Using ExecutorService we create one additional thread. It is scheduled at fixed delay so the next iteration begins only if the previews has finished. Then we invoke three methods from BLL. They return new screens, deleted screens and modified screens. Finally, collected data is passed to update methods used in observer pattern.

## **Listening for changes in CSV files**

In model when getting modified screens, we also get screens for which CSV files were changed. To get that information we need to go to Data Access Layer. There we have another thread that is running in the endless loop. If any change appears, the filepath of the modified file is added to the list that later is accessed and cleared.

## **CSV and refresh now**

When button refresh now is pressed or csv file is modified other instances of the program have to be notified as well. Then we need to set information in Database that will communicate to them that such update is required. We do that by changing the column "refresh now". We set the value to true for the screen that has to be refreshed (if we change csv file screen has to be refreshed as well).

Then there is one problem we had to overcome. If then the value of "refresh now" will be always set to true, screen in question will be updated all the time and it will be completely not functional. Taking another approach we could set the value back to initial "false", right after we read that but it's bad approach too. In this case we would send information to the



first instance of the program that will read the data from database. Other instances wouldn't be informed.

Then we found correct solution but it requires a way around. We run another thread that is scheduled with fixed delay. This delay is bigger than the delay for the listening for changes. Then each instance of the program can be informed.

```
88     Runnable runnable2 = () ->{
89         try {
90             logic.setRefreshes(); //sets to 0 |
91         } catch (BLLException e) {
92             e.printStackTrace();
93         }
94         forgetAbout.clear();
95     };
```

However, now problem isn't solved completely. We still need to avoid the situation in which we will refresh client many times (it can be irritating for the employees on the production). Then we do that in this way (snippet below). From the modified screens we delete screens that were updated very recently.

```
70         screensToRefresh.removeAll(forgetAbout);
```

We add elements to list forgetAbout when method update is called. We clear List *forgetAbout* about in runnable2 (second snippet above).

### Transaction

We use transaction for saving screen to database. We decided that it can be a good precaution from future exceptions if we save either everything or nothing.

### Try-with-Resources

In data access classes we heavily use try-with-resources. In the try statement we put only Prepared Statement or Statement, because for Connections we use Object pool pattern that will be mentioned later. In this situation closing Connections would be counterproductive.

### Loading CSV files

We noticed that CSV files can be very different depending on what data they store. That's why we decided to ask the user for some extra information, to know how they want their data to be presented. After choosing a CSV file in *Create New*, a second window appears asking the user if first row in the file contains headers, what should be the type (bar chart, line chart or table) and the title for chosen chart. After giving all the needed information we save it using *CSVInfo* class and build the actual chart / table.

## Building preview

To build a preview first, we get all Screen Elements (sections) from the database for the screen we want to show. Then, going through each section, we check what is the type of the file that would be displayed in it. Then, we create an anchor pane in one of the methods, here example of the CSV method:

```
private AnchorPane loadCSV(String filepath, CSVInfo csvInfo) {
    AnchorPane anchorPane = new AnchorPane();
    anchorPane.setPrefSize( w: 300, h: 300);
    CSVLoader.setDestinationPathCsv(Path.of(filepath));
    switch (csvInfo.getType()) {
        case TABLE -> CSVLoader.createTable(csvInfo.isHeader(), anchorPane);
        case BARCHART -> CSVLoader.createBarchart(csvInfo.isHeader(), csvInfo.getTitle(), anchorPane);
        case LINECHART -> CSVLoader.createLinechart(csvInfo.isHeader(), csvInfo.getTitle(), anchorPane);
    }
    System.out.println("loaded csv");
    return anchorPane;
}
```

Then, we add each section to the Grid Pane.

```
anchorPane.setMinSize( w: 0, h: 0);
gridPane.add(anchorPane, section.getColIndex(),
             section.getRowIndex(), section.getColSpan(), section.getRowSpan());

GridPane.setHgrow(anchorPane, Priority.SOMETIMES);
GridPane.setVgrow(anchorPane, Priority.SOMETIMES);
```

And after iteration through sections is finished, we display the grid pane.

## Assigning users to screens

We can assign multiple users to one screen, and we can add one user to more than one screen too. When the save button in the *Create new* screen view was pressed, then we would get a new window where we can choose from the list of users. Thus, the selected users or user will be assigned to the created screen.

Here, if everything went well in the save method, that would mean that we could create the screen. Then we look up the new screen ID by name from the database and we send it to the AssignUserController.

When we press the save button in the Assign window, in the btnAssign we will update the database with the new screen ID and the IDs of the selected users.

```
public void btnAssign(javafx.event.ActionEvent actionEvent) {
    List<User> selectedUsers = userTableView.getSelectionModel().getSelectedItem();

    if(!isEdit) {
        for (int i = 0; i < selectedUsers.size(); i++) {
            screenModel.saveToUsersAndScreens(screenID, selectedUsers.get(i).getID());
        }
    } else screenModel.updateAssignedUsers(screenID, selectedUsers);
}
```

## Logging in

The login contains 2 text fields (userName, and password) and a confirm button.

If an admin tries to log in and if the username plus password is correct, this admin will find himself in the admin interface. All admins will have the same interface, but in the case of users, the selected screen will appear.

But if a user wants to log in, the process is similar to the one of the admin's, but with the difference that when the password and username are correct, the selectScreen method takes effect. If this user is added to one or more screens, a Combobox will appear containing a list of these screens.

After the user writes the userName and password, the combo box will appear and they will choose one of the available screens

If the user fails to log in but only password is incorrect, we invoke shaking animation on password field. If user fails to log in and both password and log in are incorrect, we invoke shaking animation on both. Moreover, we use RequiredFieldValidator to inform a user that they forgot to provide text in one of the input fields.

## All/used screens

At the top of *Screens* view we have two toggle buttons which enable the admin to filter the list of the screens that are being displayed. "All" is just every screen that was created while "Used" are the screens with at least one user assigned to them.

```
@Override
public synchronized void initialize(URL url, ResourceBundle resourceBundle) {
    allScreens = ScreenModel.getInstance().getMainScreens();
    loadScreens(allScreens);
    for(Screen screen : allScreens) {
        if(!ScreenModel.getInstance().getUsersForScreen(screen.getId()).isEmpty()) activeScreens.add(screen);
    }
    ToggleGroup toggleGroup = new ToggleGroup();
    toggleGroup.getToggles().addAll(activeBtn, allBtn);
    activeBtn.setOnAction(event -> showActive(event));
    allBtn.setOnAction(event -> showAll(event));
}
```

At initializing the Screens window we create two lists – allScreens and activeScreens. We add every screen from database to the first one and then check if the specific screen has any users assigned to it. If yes, we add that screen to the second list.

```
private void showAll(ActionEvent event) {
    space.getChildren().clear();
    loadScreens(allScreens);
}

private void showActive(ActionEvent event) {
    space.getChildren().clear();
    loadScreens(activeScreens);
}
```

Then, by choosing one of the buttons we go to loadScreens method and show screens for one of the lists.

#### 4.4.2. Design Patterns/principles

##### Observer pattern problem statement.

We want at this same time follow MVC pattern and pass information to controller whenever it is needed (without controller asking for that).

##### Implementation

ScreenModel is an observable and instances of clients and screen views are observers.

Interface visible below is implemented by ScreenModel. The one thing that is noticeable is that we have two methods for notifying. It is caused by the fact that we actually have two kinds of observers. In another situation we need to inform screen's controller about a change and in another situation client controller has to be informed.

```
12 public interface IObservable {
13     //by default its public and abstract
14     void notifyManyObservers(List<Screen> added, List<Screen> deleted);
15     void notifySingleObservers(List<Screen> modified);
16     void notifyManyModified(List<Screen> modified);
17 }
18
```

Then we have two abstract classes. One of them is called ObserverMany and another is called ObserverSingle. They have a bit different function and content. ObserverMany is used for Screen controllers. It passes information whenever any screen was added or deleted. Then controller updates the view and user can see the change. Observer single is used for Client's Controller. It has to be updated only if the refresh button was pressed or CSV file it contains has changed.

##### Adding to observers

Controller of Screens and Client has a method to attach it to the observers. Methods are being invoked when FXML is being loaded. We access controllers using loader.getController() and then invoke that methods.

##### Method to attach Screens view

```
38 public void attachToObservers() {
39     ScreenModel.getInstance().attachManyObserver(this);
40 }
41
```

##### Method to attach client to observers

```
272 @Override
273 public void setAsObserver(Screen screen) {
274     ScreenModel.getInstance().attachSingleObserver(this);
275     setScreen(screen);
276 }
```

Client and Screens are treated as two other observers so that they are stored in two other sections and we have two other method to inform them about a change.

## Object pool pattern

### Problem statement

We often create a new connection in a try-with-resources statement. Operation is time consuming, its performed many times, however we don't need many instances at this same time.

### Solution

Being aware of existing numerous libraries that implement Object Pool pattern we decided to implement it on our way anyway. This decision wasn't driven by performance but rather by curiosity. Another aspect is that we have bigger impact on the way we implement it.

### Implementation details

We moved object creation part to another class. This class is also responsible for caching. Then whenever we need an instance of Connection we ask class responsible for managing connections. In our program this class is called *ConnectionPool*. Then when we no longer need connection, we validate it and release to the connections to use.

Method called `getConnection()` returns connection, but before it does that it checks if connection isn't longer than 30 seconds. If yes it is invalidated and we get new connection.

We can't add connection to try-with-resources anymore because then it would be closed after execution and we would lose all the point of using object pool pattern. Then we create a new connection, we pass it to the method and then after method executes we release connection back to the pool. If for some reason we won't be able to establish connection, custom exception will be thrown.

```
151         @Override
152         public void save(Screen screen, List<ScreenElement> screenElements, List<User> usersList) throws DALException {
153             Connection connection = null;
154             try {
155                 connection = connectionPool.getConnection();
156                 screenDAO.save(screen, screenElements, usersList, connection);
157             } catch (SQLException throwables) {
158                 throw new DALException("Couldn't establish connection");
159             }
160             finally {
161                 if(connectionPool.isValid(connection))
162                     connectionPool.releaseConnection(connection);
163             }
164         }
```

### Checked exceptions

We have very radically used checked exceptions. They were wrapped in custom exceptions that corresponded to the layer and propagated all above the layers. Finally, they appeared in the class in GUI that could explain exception in the best way.

However, we made one very deliberate exception to our general way of handling exceptions. Sometimes there is no need to inform the user with readable message. We may want to just handle exception silently. This is the solution we implemented for one of the methods in *ConnectionPool* class (snippet below). If the connection is invalid and for some reason we can't get information about its state, we just discard that and do nothing more about it

```

92         @Override
93         public boolean isValid(Connection connection) {
94             if(connection == null) {
95                 return false;
96             }
97             try {
98                 return !connection.isClosed();
99             }
100             catch(SQLException se) {
101                 return false;
102             }
103         }
104     }

```

#### 4.4.3. Unit test

Test checking if a method *getNewScreens* in *detectOtherScreens* works properly. It's one of the three methods that make a core of our synchronization functionality.

```

@Test
void getNewScreens() {
    DetectOtherScreens detectOtherScreens = new DetectOtherScreens();
    List<Screen> newScreens = new ArrayList<>();
    ObservableList<Screen> oldScreens = FXCollections.observableArrayList();

    Screen screen1 = new Screen( id: 1, name: "aa", refreshTime: 5);
    Screen screen2 = new Screen( id: 2, name: "bb", refreshTime: 4);
    Screen screen3 = new Screen( id: 3, name: "cc", refreshTime: 5);
    Screen screen4 = new Screen( id: 4, name: "aa", refreshTime: 3);

    newScreens.add(screen1);
    newScreens.add(screen2);
    newScreens.add(screen3);
    newScreens.add(screen4);

    oldScreens.add(screen2);
    oldScreens.add(screen3);

    List<Screen> expected = new ArrayList<>();
    expected.add(screen1);
    expected.add(screen4);

    List<Screen> actual = detectOtherScreens.getNewScreens(newScreens, oldScreens);

    Assert.assertEquals(expected, actual);
}

```

## 4.5. Sprint Review

We believe our second sprint review went well. We finished all the user stories which we presented to the client. We received positive feedback with only two small things to improve.

## 4.6. Sprint Retrospective

<b>Things that went well:</b>
We managed to finish all the planned tasks
Communication in a team and dividing the work
<b>Things to improve:</b>
We could have asked for help earlier rather than try to deal with problems on our own

## 5. Conclusion

Overall, the whole process was a great pleasure and a valuable learning experience, as we had the opportunity to use the SCRUM with real customers. Our group has managed to utilize all aspects of SCRUM. During the implementation process the aim was to create the product following and fulfilling the customer's needs and expectations. In the first sprint, the team mastered that we must break down large tasks into smaller ones and prioritize them effectively to be aware of the potential upcoming risks. This was the first time when we get real customers, and we must create a program based on the order. The group took the task seriously and wanted to create a program that would cover the customers' requests. Therefore, between the first sprint and the first scrum meeting, we tried to perform tasks that best show how we interpreted the task and how we imagined it. In the second sprint, we implement every feedback from the customers and corrected the first sprint's mistakes. We decided we will create two sprints until the next scrum meeting. Because we had 3 weeks until that, and we could not estimate what we put in the program. At the end of the second sprint, we summed up the missing parts and problems for the team to correct in the third sprint. During the third sprint, a team realized that there were tasks we couldn't complete on the time constraints, so we tried to complete the more important tasks. During the second scrum meeting, the team received good feedback, confirming that the ideas for the team's work over the past 4 weeks have been good. After that, the team decided to complete the program based on the requests received during the second scrum meeting and fix bugs. After that, the whole team shifted to do the report writing. Aware of our mistakes we are all proud of the work we have done and look forward to completing the next project even better.

To sum it up, the group is satisfied with the product and feels that everyone has had the opportunity to develop in areas they were not strong in before and a greater knowledge of Java. Once again, the opportunity to work with a real customer, and experience how the

customer's wishes change through the working process as the program develops was a great real-life experience for us.

## 6. References

<https://www.arla.dk/om-arla/landmandsejet/>

Bhatti, M. & Awan, Hassan & Razaq, Z.. (2014). The key performance indicators (KPIs) and their impact on overall organizational performance. Quality & Quantity. 48. 10.1007/s11135-013-9945-y.

Moodle: EASV 1st semester 2021 EXAM PROJECT ppt. Date of uploading: 26.04.2021.

<https://www.scrum.org/resources/blog/10-tips-product-owners-product-vision>

[https://www.mindtools.com/pages/article/newPPM\\_07.htm](https://www.mindtools.com/pages/article/newPPM_07.htm)

<https://coderslegacy.com/java/jfreechart-with-javafx/>

<https://www.jfree.org/jfreechart/>

<http://tutorials.jenkov.com/javafx/filechooser.html>

<https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html>

<https://www.baeldung.com/java-check-string-number>

<https://github.com/jfree/jfree->

<fxdemos/blob/master/src/main/java/org/jfree/fx/demo/FXGraphics2DDemo1.java>

<https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/>



## 7. Appendices

### APPENDIX A

#### **Product Vision Board**

##### **Vision**

By extending our own limitations and building on the programming knowledge we have gained so far, the goal is to create a desktop application for Esbjerg Dairy Center, that reduces the number of monitors used in the production, and instead, multiple screens can be displayed on a single monitor. This results in making the work of users and admins easier.

##### **Target group**

The target customer can be divided into two groups. The first one is the group of employees who participate in the production at Esbjerg Dairy Center. The goal is to provide them with an easier overview of the KPIs. The user view of the program would serve their needs. The second one is every admin employed by Arla with access to the admin view and can modify the stored data by adding, editing, deleting, refreshing, etc.

##### **Needs**

The product aims to remove the unnecessary number of monitors used in the production by using dynamic grids on multiple screens. Providing an easy-to-use user view that enables the employees in the production to find the relevant KPIs and simple admin view, which allows the ones with access to make changes in the program. The program would be used internally in Esbjerg Dairy Center's everyday production.

##### **Product**

The product itself is a desktop application that would make easier the work of the employees in dairy production and the admins behind the scene. The design strives to be clean and simple, using the colour schemes of Arla. The program is going to be easy to use both as a user and an admin, and it would work without Internet access too. It is also capable of displaying multiple formats on screens.

##### **Business goals**

Decrease the time spent looking up to several monitors in order to find the information you are looking for as a user. Minimizing the number of monitors in use would help to save additional costs for the company. It will increase better transparency regarding the display of the KPIs and that might increase productivity too. Admins from Arla would have a better-distributed workload and could track/make changes way easier in the program.

## APPENDIX B

### Risk analysis Table

The table below shows an overview of the identified potential risks as part of the project's risk analysis. The team used the Probability and Impact scores to rate the different possible factors properly. The factors are rated on a scale between 1-5. Key: 1 means less likely with the lowest impact on the outcome; and 5 means the most likely with the highest impact on the project's outcome.

Identified risk	Probability Score	Resource/Time/ /Scope	Impact Score	Mitigation
Team sickness/injuries	2	Scope	2	Reduce scope/ task redelegation
Team communication issues	2	All	3	Internal consultation/ Daily meetings
External factors	1	Scope	2	Reduce scope/reorganise the tasks
Code stuck	3	All	4	Reduce scope
Avoiding/asking help too late	2	All	4	Consult with team members/teachers/t utors
Deadline delay	3	Time	5	
Quality issues	1	All	1	3 layers architecture, DoD
Technical difficulties	1	Time	2	Ask help form EASV
Loss of client	3	All	5	Ask questions from the clients/ Listen the feedbacks proactively

## APPENDIX C

### Working Agreement

#### Working hours and time schedule

We understand that members have other duties and responsibilities other than this project. Then we create a schedule of working hours and time when we are available for the contact. Availability for the contact: 9 – 15

#### Daily meetings

We meet every day for short meetings. The organizational part shouldn't take more than 15 minutes. If agreed other members can stay longer to solve some problem but it's not compulsory. We agree on the time of the meeting in advance of one day ahead.

- We always show up, collaborate and try to hold a meeting as short as possible
- If somebody can't show up they must inform the other group members before the scheduled meeting

#### General behavior

The objective of this project is both to practice coding skills and to practice collaboration and teamwork when developing coding skills together.

Kamila Potanik

Leif  
Caelen

DA'SZLOL NAY

## APPENDIX D

### Log in information

Username	Password	isAdmin
andi1	asd123	false
paki	asd1234	true
Magic	123asd	false
pala	lol123	false
soma2	soma2	false
toka	toka	false
zsemle	asd1235	true
asd	asd	false

## APPENDIX E

### **Note from the first group meeting:**

Client window:

- client cant resize the windows
  - within the window client can zoom in and zoom out
  - static data
- show CSV, PDF, webpage given the URI, Excel(.xlsx) <- after meeting do a reaserch how to open in in java etc

Admin window

- somewhere they have a list of users(setups) and they can choose which one they wanna modify now
- after you choose user -> select how many windows -> select for each window what it shows
- create one template (then we can add moree)
- admin should have a view with a list with all active set ups (in the scrolling list)
- functionality: set refreshing for each active window and have one button refresh which instantly refreshes

1st idea -> drag and drop like windows (for making the file appear in the program) The best idea

How to store information for the set up: important

- UserID/Name name can be an id but lets think about ti later
- how windows are divided
- which data type is in each of them
- save file names for each window

-----

conclusion

- we need to do prototyping
- db design is still unclear <-hard part
- that fancy drag and drop can be easy or extremely hard
- refresh information on connections confugurations in JDBC

Plan:

- think about the prototype for admin view
- that fancy drag and drop can be easy or extremely hard; all of us do a reaserch

Database: - later. prototypes first