Arla - Less Screens

Exam project

2nd Semester

Kjell Schoke & Mikkel L. Mouridsen

Jeppe Moritz Led
Software Construction, Software Design &
IT and Organizations

26.04.2020 - 01.06.2021

1 Preface

Contents

2 Introduction

2.1 Background

2.2 Problem definition

How can we as computer scientists, create an Java application that implements JavaFX, that solves the problem of having too many screens and avoid and still give the users a good overview of all the KPI's without being too cluttered and user-hostile.

In order for Arla to achieve key business objectives, they use so-called Key Performance Indicators, or KPI's, to track their production and thereby maximize the products quality and quantity. These KPI's are presented in numerous forms of data that needs to be available to the workers, in preparation for an analysis of their work. This data is presented in the production environment on a great number of screens, which are connected to the server via an RDP connection that is controlled by the it-department. The program is running on the it-departments server and sending the live image of the running program to the clients which are connected to the screens. This is done via RDP (Remote Desktop).

2.3 Product vision

In order to avoid any additional obstruction during the employees' analysis of the key performance indicators at Arla, we created an application which gives the users an easy overview of their departments respective KPIs. One of the main tasks of this program, is to reduce the amount of monitors used to display important charts, data charts, internet websites or PDF files.

It is difficult to know for sure, which kind of people are going to use this application on a daily basis, we can of course make some broad assumptions to narrow the target group a bit and thereby create personas. These personas can help us in making design choices, for the program, and lessen, or increasing the learning curve of the application depending on the created personas. Hubspots persona tool [1] was used to create the personas

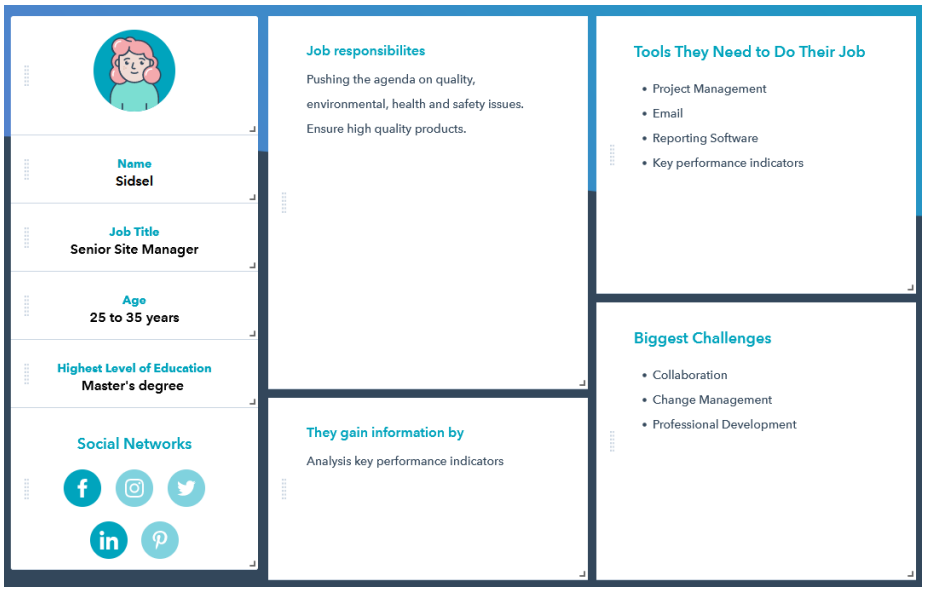

Figure 1: Arla Production Employee - Persona

Since the application will be employed both in the production management and for the production workers, we will need to create a persona for both of these types. The requirements for these job positions is varied since it highly depends on which tasks you are assigned to. For example, a regular person working at the production line does not require the same degree or experience as someone working at CO2 emission control. For our production line persona we will take a student worker, who currently is in the process of getting a degree.

Oscar is a typical college student, currently working as a production line worker at Arla, he packages products and assesses the package quality. He works closely

[1] https://www.hubspot.com/make-my-persona

together with his coworkers and reports any issues for the day at the end of his shift. See figure 9 for the image of the persona. With computer applications playing a bigger role in the lives of people, especially for younger people like Oscar, we know that he will probably not be that confused by a more complex program. See figure 9 for an image of the persona.

Figure 1 depicts a typical person that works at Arlas production management, her name is Sidsel and works as the Senior Site Manager, who ensures the agenda on quality, environmental, health and safety issues. This persona is based on one of the people who actually work at Arla, see [1].

With this persona we can assume that the employees who will use this application are very skilled in their respective occupation, but not necessarily the fastest learned when it comes to computer programs. With this information we know that the application should be build simple to avoid any confusion with the normal users (non administrators.) Administrators will of course be more well versed with computer programs, and thereby not be too confused by a user interface which features more advanced elements.

From the kick-off meeting with the product owners we know that the Java application must be able to visualize the key performance indicators with different methods of data visualization, as a minimum the program should be able to show:

- Websites (URLs)
- Pie charts
- Bar charts
- Line charts
- PDF
- Raw CSV data
- Raw Excel data (.xlsx)

In addition to this the application also has other requirements which are set in place in order to make the program it self more usable in its day-to-day usage. A tile (in which data can be presented) should have a full screen functionality implemented, so any type of data can be viewed in a higher resolution, this can for example come in handy, when the production department is holding a meeting and you want to show data in a more presentable fashion. Furthermore, the application should also handle user logins, administrate users and assign dashboard configurations (or dashboard layouts) to users.

In order to create user backlog items for the SCRUM part of the project, we created four epic user stories:

1. As an admin, I want to be able to manage all users in the system
2. As an admin, I want to be able to make different dashboard configurations that I can assign to users
3. As a user, I want to be able to access the same dashboard every time I use the system
4. As an admin, I want to be able to add different sources of data to the different dashboard configurations

2.4 Strategic analysis

2.4.1 Stakeholder analysis

The stakeholder analysis helps us identify and group the stakeholders. Stakeholders are "individuals (or groups) that can either impact the success and execution of a product or are impacted by a product." [2]. of the project. In this case, we are assessing the stakeholders who are closely bound to the project, we could, of course include a lot more, but this would be beyond the scope of this project. We chose four main stakeholders,

1. Internal: Developers
2. External: Production Management (This includes IT-personal responsible for the implementation of the product), Production Worker (Product line workers etc.) and Teachers.

The internal and external stakeholder are placed into a power & interest matrix, and their position in this matrix was determined by their power (or influence) and interest in the project.

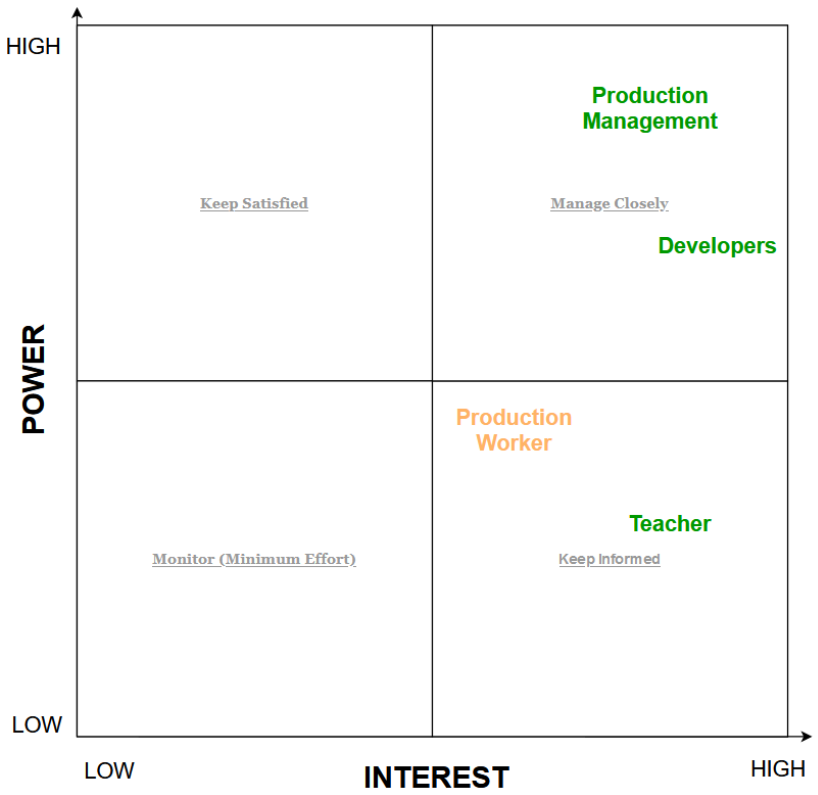

Figure 2: Stakeholder analysis

The *production workers* were evaluated to be a bit of risk to the project, since they are the stakeholder that is the most affected by the change to a new application for the analysis of key performance indicators. The product management will probably not be that troubled with the change of systems, since they wont be working with it in an environment where they also have to keep track of other things in a possibly stressed situation (e.g. production line workers.) Even though the production workers are not identified to be under the category *Manage Closely*, they still do, to some extent, have an influence on a possible system reversal if they are dissatisfied with the product.

2.4.2 Risk analysis

Additionally we also made an risk analysis in order to easily combat any of the most reasonable risk that can occur during the development of a project.

Table 1: Risk analysis

Risk source	Risk description
Illness & Injury	Whenever a team member gets sick, and thereby is unable to work, they have to inform the other team mate before the next daily scrum meeting.
Electronic failure	When a team members computer fails and thereby is unable to work, they have to inform the other team mate before the next daily scrum meeting, in addition to this, you have to push all your work on GitHub to you current work branch.
A task taking longer than expected	When a task already is taking longer time than expected SCRUM has to be updated accordingly, if the task is a key task, contact your team-mate and discuss possible solutions to this problem.
Poor quality product	When, during a daily SCRUM meeting, the other team member is dissatisfied with the work on a task of another team mate, they will have to discuss this and together, find a solution which satisfies both parties
Conflicts within the team	When a team member is dissatisfied with the other team mate (e.g. lack of work input etc.) they have to discuss this, and find a way to manage this problem.

Green = Low Risk, Orange = Medium Risk, Red = High Risk.

3 Pre-Game

3.1 Project organization

3.1.1 SCRUM process

To organize and manage this project, the agile framework SCRUM was used. SCRUM works in an iterative and incremental manner, which makes it easy for the product owner, to make eventual changes throughout the development of the product. Normally a sprint (time frame where work is done) lasts from one to four weeks and every sprint with a sprint review with the product owner, stakeholders, scrum master and the development team [4]. In our case, the sprints are different lengths, though this does not have an effect on the product outcome, in addition we also had five days at the end of the second sprint review meeting, which gave us time to finish the report and do code refactoring. On figure 3 is an illustrative overview of the SCRUM framework.

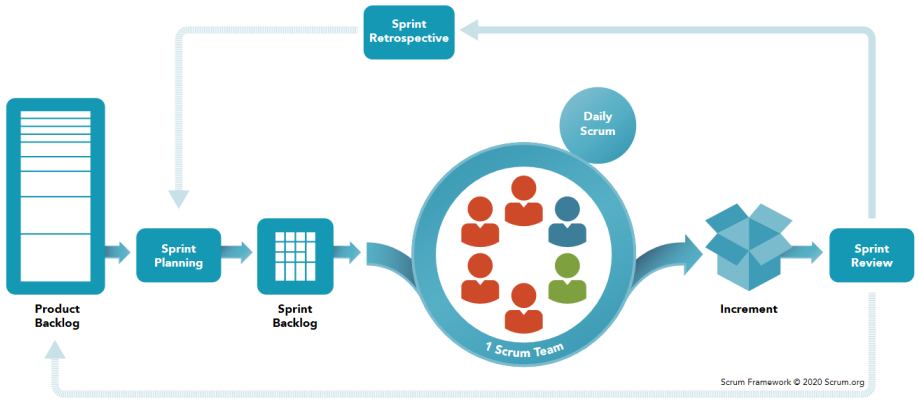

Figure 3: The SCRUM framework [3]

In our case the SCRUM process started with a kick-off meeting with the key stakeholders at Arla, where they presented us their problem and their wanted solution for this problem. Afterwards, Mikkel, the assigned product owner for this project, constructed the product backlog items for the whole project based on the stakeholders expectations. We then planned for the upcoming sprint, e.g. which tasks are assigned to who or who design the graphical user interface, or added the newly features to as product backlog items after a sprint review. The daily SCRUM meetings were held at the start of each work day for fifteen minutes, where we discussed what has been done and what will be worked on until the next daily meeting. After the sprint has finished a sprint review meeting was held with the key stakeholders at Arla and subsequently a sprint retrospective was established. After the sprint retrospective a new iteration will begin with a new sprint.

As our team is a very small (two members), we are forced to assign the SCRUM roles a bit different. We decided that in the first part of the project (the first SCRUM sprint) Mikkel will be assigned as the SCRUM master, he will be responsible for the specification of the definition of done, and make sure that both of us will adhere to it, and assuring that the SCRUM process will be applied in a correctly manner. Mikkel will also act as the Product Owner for the whole projects time span, he will make sure that the product backlog items are written and added to scrum wise, though this work is not exclusive to the product owner, since both Mikkel and Kjell somewhat act as the Product Owner for the project.

After the first sprint review meeting, the SCRUM Master position will be handed over to Kjell, he will have the same responsibilities. Both team members will, of course, also act as part of the development team, further role assignments are discussed in section 3.1.3.

3.1.2 Definition of done

In order to concisely determine if a part of the product is ready for the next sprint review, or the whole project is finished as a whole, we denoted a definition of done, which consists of a list of requirements which have to be met.

The first definition of done is only meant for finishing a user story:

1. Expected functionality is implemented.
2. Project builds without issues.
3. Basic testing of the implementations are made (In program testing.)
4. SOLID principles are applied.
5. Code is commented with Javadocs.

The second definition of done is meant for the completion of the two sprints (end of exam project):

1. The first definition of done is applied to every user story.
2. All unit tests are green.
3. Every To-do note is resolved

3.1.3 Development role assignments

As you normally would in a scrum based environment, we assigned the tasks by our own skill level, Mikkel, who in general is very proficient in all the types of programmatic wise skills is responsible for the mostly logical part of the program (business logic layer) and for the implementation of the program design patterns. Kjell on the other hand, is responsible for the design of the graphical user interface, and its implementation. Kjell is also responsible for the design of the Microsoft SQL Server and its implementation in the program.

3.1.4 Team agreement

In a nutshell, team agreements assist team members understand what is expected of them in terms of work and team culture, and they allow people to hold one other responsible. This agreement can include a wide range of topics, including how members will collaborate, make decisions, communicate, exchange information, and support one another. The rules for how members will and should behave with one another are clearly outlined in the team agreement. The team agreement for the Arla project is as follows:

Working time

- Working hours are 12 pm to 18 pm. (Can be moved if a team mate is obstructed from attending.)
- Breaks are taken whenever you feel the need.
- 10 minute standup meetings every day.

Penalties

- If you arrive late, you will pay for a beer for every hour you are late. This only applies if you arrive late without telling your team mate, e.g. when you're sick or in any other way unable to work.

Standup meetings

- If you arrive late, you will pay for a beer for every hour you are late. This only applies if you arrive late without telling your team mate, e.g. when you're sick or in any other way unable to work.
- The presentation will include showing the features added, and an explanation on how the feature is implemented with showing code et cetera.

Signature - 27. May 2021

Signature - 27. May 2021

3.1.5 Use of software

In order to assure a smooth development of the project, the team used multiple types of software to either communicate and administrate task, visualize product prototypes and program the product itself. Scrumwise, Discord, Figma, Intellij IDEA, SQL Server Management Studio, GitHub, Diagrams and Visual Paradigm was used during the development.

Discord was used to communicate between team members, on occasion also to ask questions to a teacher to clear confusion. Discord was also used to administrate tasks (SCRUM and report tasks) to one another. Intellij IDEA was used to code the product, Microsoft SQL Server Management Studio (SSMS) was used in order to communicate with the schools database server. The schools SQL server is only accesible if you are on the same network as the server, the connection was able to be made with the schools VPN, GlobalProtect, because of the corona pandemic, this was of course the only way we could access the server, since it was prohibited to enter the school for the most part of the exam project. Scrumwise was used as our way to organize SCRUM related matter, such as making burn down charts or marking our progress of the assigned tasks. To version control our application we used GitHub where we made branches based upon the assigned tasks, once the functionality was fully implemented and tested is was merged with the main development branch and the old task branch was deleted.

Visual Paradigm was used as the way to make UML based class diagrams and Figma was used for the prototyping part of the development, where we made mock ups of our graphical user interface. In addition, Diagrams (also know as draw.io) was our drawing tool to make other types of diagrams, e.g. the stakeholder analysis.

3.2 Project schedule

The different report sections in the schedule table, table 2, are meant as the start of clue writing, e.g. the clue for "problem definition" will be written somewhere in the span of week 17 to 18, but this does not mean that the whole section has to be finished in that time span.

Table 2: Overall project schedule

Activity	Date
Week 17 to 18	
Kick-off meeting, start of first SCRUM sprint	Monday 26/04, 09:00
Problem definition	
Product vision	
Strategic analysis	
Project organization	
Team agreement & definition of done	
Brainstorming of the graphical user interface	
Creating the products code base	
First SCRUM review meeting	**Friday 07/05, 14:10**
Week 18 to 21	
Start of Second SCRUM sprint	Saturday 08/05
Initial Product Backlog	
Prototype Mockup	
Coding	
Report Writing	
Second SCRUM review meeting	**Thursday 27/05, 10:40**
Week 21 to 22	
Code refactoring	
Report writing	
Report and program hand-in	***Tuesday 01/06, 14:00***

3.3 Initial product backlog

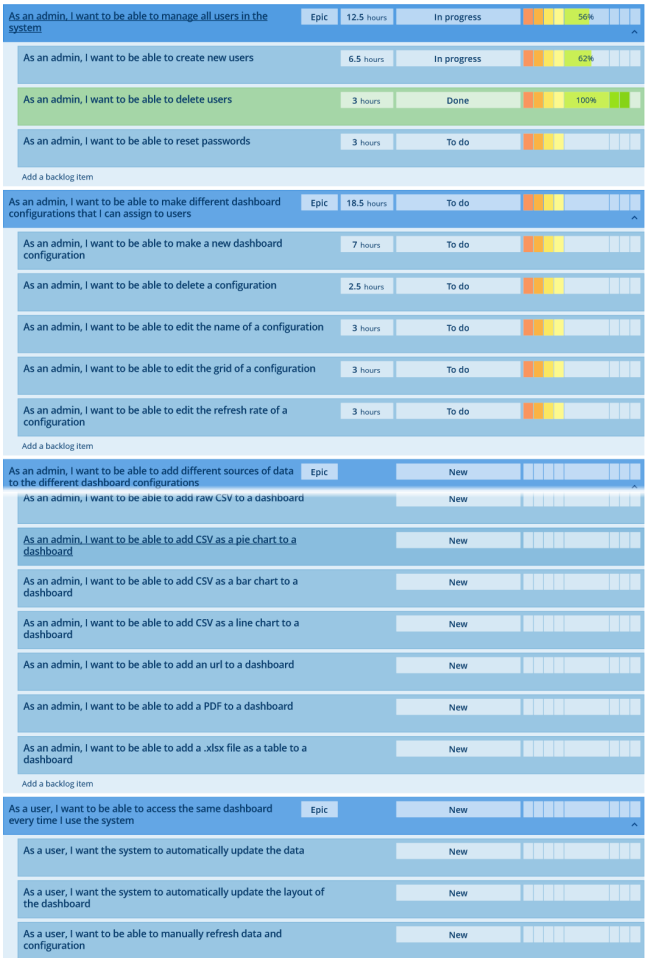

Figure 4: Initial product backlog

On figure 4 is an image of the initial product backlog.

3.4 Architecture

We decided to structured our application as a 3-layered architecture. The 3 layers consists of a presentation layer, business logic layer and a data access layer. The important this with this structure is that all the layers are loosely coupled. We do this by using interface, but we also make it so that the layers only know about one other layer, or none at all.

3.4.1 Presentation Layer

This layer handles all logic related to presenting an application to the user. This layer only knows about the business logic layer, and gets all data through that layer. We used JavaFX to make the user interface, and as that is closely tied to MVC, we used that pattern to make our User Interface. The MVC pattern will be explained later in this section.

3.4.2 Business Logic Layer

The business logic handles all logic, and/or processing that is done to our business entities. It also calls the data access layer, to get data from the database as business entities. The only other layer that this layer knows about is the data access layer.

In previous projects we had trouble with having persistent data across the application. To avoid this issue we decided that we wanted to implement some kind of system, that could keep track of all business entities. We searched a bit for a way to do this, and found a design pattern called the repository pattern. The repository classes would expose simple methods for adding getting and deleting business entities. It is kind of like an in memory representation of our data, which comes from our data source. The repository classes are not responsible for handling interactions with the database, that is delegated to the data access objects in the data access layer.

On figure 5, is the first draft of how we wanted to structure the repository classes. In the figure we use the repository of a user as an example. We have a base interface that all the repository implementations are going to use, which has all basic methods for getting, adding and removing business entities. We then make the different repositories available through a facade, which is also a singleton. The reason for making the facade a singleton, is so that we can make sure there is only one instance of all repositories. Technically there could be more than one of each repository, as the repositories themselves are not a singleton, but as we only access the repositories through the facade this should not be a problem This whole design is later extended a bit, to get cleaner code. This will be discussed later on.

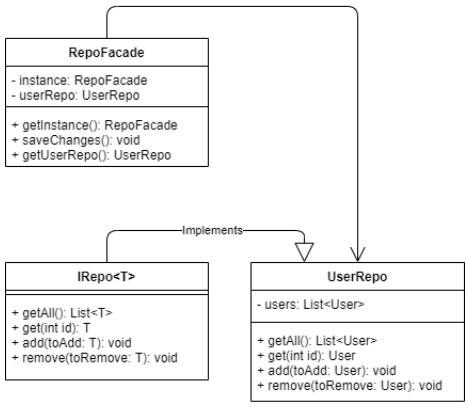

Figure 5: Initial repository design

3.4.3 Data Access Layer

The Data access layer is responsible for querying data from the database, and writing new and/or change data. The data access layer knows nothing of the other layers, but it knows about the data source. In our case this is a Microsoft SQL database server. But it could also have been something like a file or a REST API. So to put it simple this layer is responsible for all CRUD operations.

We decided to go with simple objects that are responsible for creating, reading, updating and deleting from our database. We call these objects data access objects.

3.4.4 Model View Controller (MVC)

Model View Controller is a design pattern that consists of three parts. Models, views and controllers.

Model

Models is all the logic that does stuff with our business entities. This is everything from our classes that represent data between the view and controllers (In our case these classes all end with Model in their name) to the classes that handles different operations on our business entities. [7]

View

This is what handles showing our user interfaces. Usually this is all of the code which handles showing the UI. [7] But, as we use JavaFX, this part of the pattern is our fxml files. Which is a special version of XML, that is used to describe what our user interfaces look like. All the code that handles showing data is already made, and we only need to describe what our view should look like.

Controller

Controllers is the final part of this pattern, and it is like the glue that binds models and views together. Making communication between the two possible, without them knowing about each other. [7]

3.5 Preliminary usability test

4 Sprint 1

4.1 Sprint planning

This sprint will mostly be dedicated to the setup of the program, we will build the code base for the three layered architecture (presentation, logic and data layer) and do some of the implementation needed so we have something to present at the first sprint review. Here we ordered the product backlog items with the most important at the top, and the lesser important at the bottom. All of the items are of course important, but since we need a basis for the further development of the program, we put the necessary features at the top of the list.

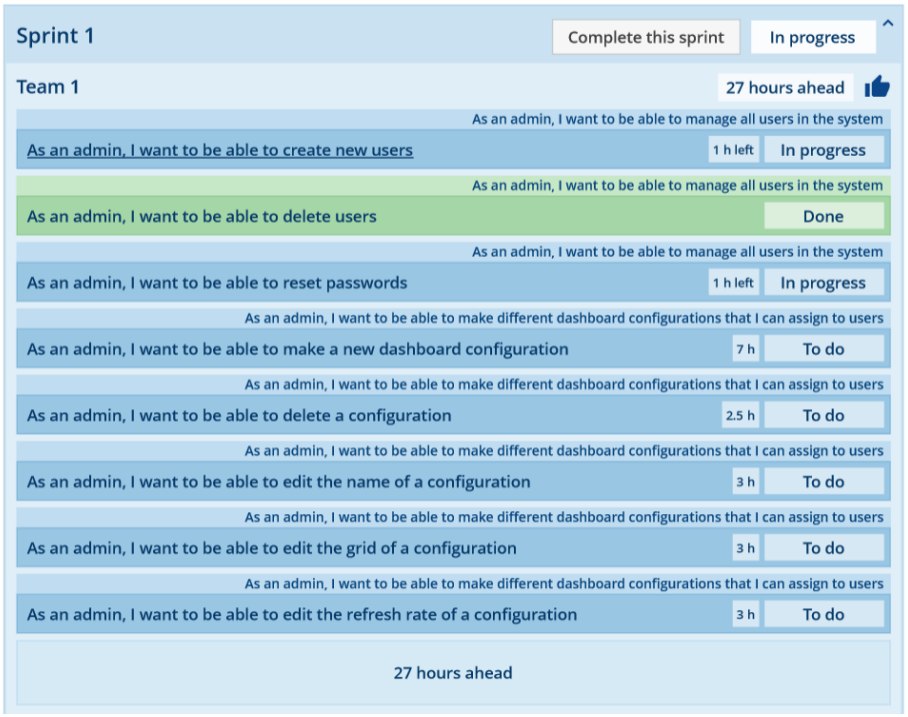

Figure 6: First sprint product backlog items

4.2 Daily meetings

4.3 GUI

4.4 Implementation

During this sprint, a lot of the basic features was implemented. This includes a login, that will direct a user to their version of the application. This is based on what privileges the user has, if the user is an admin or not. The admin panel, where admins has the ability to administer users and dashboards, was also implemented. The user dashboard was also implemented so that normal users can login and be shown their assigned dashboard.

4.4.1 Code examples

The first thing we implemented, was the repository system. We started by defining an interface, that all repository classes can implement. We did this so that we can make sure all repositories has the same base functionality.

IRepo.java

```
1     List<T> getAll();
2     T get(int id);
3     void add(T toAdd);
4     void remove(T toRemove);
5     void saveAllChanges() throws DataAccessError;
```

We then wanted repositories to be able to notify to selected listeners, that the repository has changed. We accomplished this by making an abstract class, that implements the IRepo interface. This class has a method for subscribing, to become a listener, and a method for notifying the listeners that a change has occured in the repo.

ObservableRepo.java

```
1  public abstract class ObservableRepo<T> implements
       IRepo<T> {
2
3      private List<IRepoListener> listeners = new
           ArrayList<>();
4
5      public void subscribe(IRepoListener listener) {
6          listeners.add(listener);
7      }
8
9      public void notifyRepoChange() {
10         for (IRepoListener l : listeners) {
11             l.userRepoChanged(this);
12         }
13     }
14
15 }
```

This enables other classes to register themselves as listeners, if they implement the IRepoListener interface. They will then get notified when a change in the repo has occured.

We can then inherit from this abstract class, if we want a repository to have this base functionality. Here is an example of one of the classes that inherits ObservableRepo. This repositories keeps track of all new and deleted users, and also when changes has been made to the different business entities. Then when the saveChanges method is called, it will call the data access layer, so that new, deleted and changes can be reflected in the database.

UserRepo.java

```
1  public class UserRepo extends ObservableRepo<User> {
2
3      private List<User> users = new ArrayList<>();
4      private List<User> newUsers = new ArrayList<>();
5      private List<User> deletedUsers = new ArrayList<>();
6
7      private IDataAccess<User> userDataAccess = new
           UserDataAccess();
8
9
10     private LocalDateTime lastUpdated =
           LocalDateTime.now();
11
12     public UserRepo() throws DataAccessError {
13         users.addAll(userDataAccess.readAll());
14     }
15
16     @Override
17     public List<User> getAll() {
18         return users;
19     }
20
21     @Override
22     public User get(int id) {
23         for (User u : users) {
24             if (u.getId() == id) {
25                 return u;
26             }
27         }
28         return null;
29     }
30
31     @Override
32     public void add(User toAdd) {
```

```
33          users.add(toAdd);
34          if (toAdd.getId() == -1) {
35              newUsers.add(toAdd);
36          }
37          notifyRepoChange();
38      }
39
40      @Override
41      public void remove(User toRemove) {
42          users.remove(toRemove);
43          if (toRemove.getId() != -1) {
44              deletedUsers.add(toRemove);
45          }
46          notifyRepoChange();
47      }
48
49      @Override
50      public void saveAllChanges() throws DataAccessError {
51          try {
52              for (User u : users) {
53                  if
                        (u.getLastUpdated().isAfter(lastUpdated))
                        {
54                       userDataAccess.update(u);
55                  }
56              }
57              lastUpdated = LocalDateTime.now();
58
59              for (User u : newUsers) {
60                  userDataAccess.create(u);
61              }
62              for (User u : deletedUsers) {
63                  userDataAccess.delete(u);
64              }
65              newUsers.clear();
66              deletedUsers.clear();
67          } catch (DataAccessError e) {
68              throw e;
69          }
70          notifyRepoChange();
71      }
72
73      public User get(String username) {
74          for (User u : users) {
75              if (u.getUsername().equals(username)) {
76                  return u;
```

```
77              }
78          }
79          return null;
80      }
81  }
```

We also implemented the data access objects, which are going to do all the database transactions. In the same way as the repositories, we also made a base interface for all data access objects to implement.

IDataAccess.java

```
1  public interface IDataAccess<T> {
2      void create(T toCreate) throws DataAccessError;
3      List<T> readAll() throws DataAccessError;
4      T read(int id);
5      void updateAll(List<T> toUpdate);
6      void update(T toUpdate) throws DataAccessError;
7      void delete(T toDelete) throws DataAccessError;
8
9  }
```

As shown on the listing above, the IDataAccess interfaces defines methods for creating, reading, updating and deleting (CRUD). We can then use this interface whenever we want to create a new data access object.

UserDataAccess.java

```
1  public class UserDataAccess implements IDataAccess<User>
       {
2
3      private final String CREATE_SQL = "INSERT INTO Users
           (Username, Password, IsAdmin, ConfigID) VALUES
           (?, ?, ?, ?);";
4      private final String DELETE_SQL = "DELETE FROM Users
           WHERE ID=?;";
5      private final String SELECT_ALL_SQL = "SELECT * FROM
           Users;";
6      private final String UPDATE_SQL = "UPDATE Users SET
           Username = ?, Password = ?, IsAdmin = ?, ConfigID
           = ? WHERE ID = ?;";
7
8      private DBConnector dbConnector = new DBConnector();
9
10     @Override
11     public void create(User toCreate) throws
           DataAccessError {
```

```
12          try (Connection conn =
                dbConnector.getConnection()) {
13              PreparedStatement statement =
                    conn.prepareStatement(CREATE_SQL,
                    Statement.RETURN_GENERATED_KEYS);
14              statement.setString(1,
                    toCreate.getUsername());
15              statement.setString(2,
                    toCreate.getPassword());
16              statement.setBoolean(3, toCreate.isAdmin());
17              statement.setInt(4, toCreate.getConfigID());
18
19              statement.execute();
20
21              ResultSet keys =
                    statement.getGeneratedKeys();
22              if (keys.next()) {
23                  toCreate.setId(keys.getInt(1));
24              }
25          } catch (SQLException e) {
26              throw new DataAccessError("Could not create
                    user: " + toCreate + " in database", e);
27          }
28      }
29
30      (...)
31
32  }
```

In the listing above, is showed how the create method for the User data access object has been implemented. It starts by getting the connection to the database, and the creates a prepared statement. The reason why we use prepared statements, is so that we can protect against SQL injection. We then execute the statement, after setting the various values of the statement. Lastly we get the newly generated ID of the user, and update the business entity with the ID.

4.4.2 Design Patterns & Principles

As mentioned above we use 3 layered architecture in this project, seperating UI, business and data access logic.

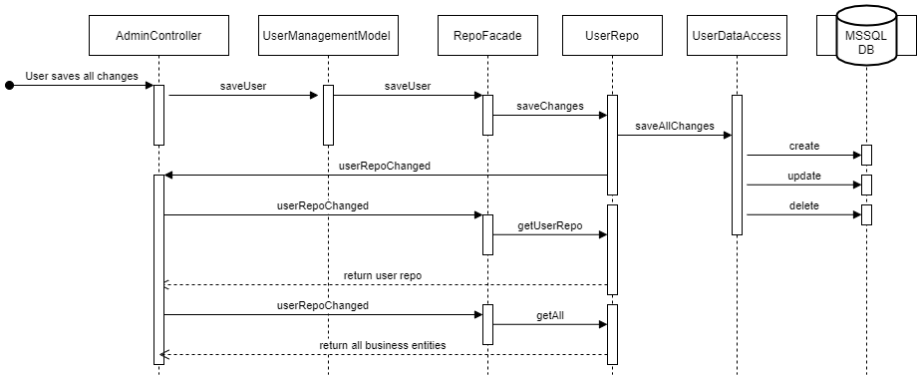

Figure 7: Sequence Diagram of saving all changes

To see how data flows through the 3 different layers of the application, we can look at the process of saving changes made to the user repository. We can start all the way from the interaction the user makes, to data being written to the database.

As shown on the sequence diagram on figure 7, when the user clicks on the save button the saveUser method will be called in the AdminController class. This will then call the saveUser method on the UserManagmentModel class. This class will then call the RepoFacade, which means we not transition over to the business logic layer. This will then call the saveAllChanges method on the UserRepo which then calls the different methods for creating, updating and deleting in the Data Access layer. These methods will then interact with the database. When all of that is done the UserRepo will notify all its listeners, that there has been a change in the repository. As the AdminController is a listener, the userRepoChanged method will be called. This method then calls the RepoFace to get a reference to the UserRepo, and then it can get all of the business entities from the repository.

As mentioned above there is a class called RepoFacade. This is part of a design pattern used, called the facade pattern. The intent of this pattern is to provide a interface that can represent an entire subsystem. [5] At the current stage of the software, the facade is only exposing ways to retrieve the different repositories and saving changes in all repositories. However, as the software grows this might be extended. To see how this is structured, we can take a look back at figure 5.

The factory pattern was also used multiple places in this first iteration of the software. The intent of this pattern is to provide an interface than is responsi-

ble for creating different objects. [6] We use this pattern mainly for two things. Instantiating cells for the dashboard grid, and creating different kinds of Dialogs.

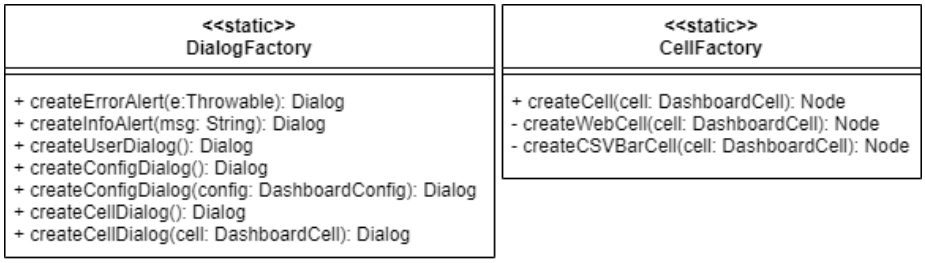

Figure 8: Factory Pattern Implementation

On figure 8, the two implementations of the factory pattern are shown. The first one being the DialogFactory class, where there are multiple methods for creating different kinds of dialogs. The second is the CellFactory class, where we only expose one method so and the class figures out what type to create based on the dashboard that is given as a parameter. Both of these classes have been made static, so that they can easily be access from everywhere in the code. However, it is only used in the UI layer of the code.

4.5 Sprint Review

4.6 Sprint Retrospective

5 Sprint 2

5.1 Sprint planning

5.2 Daily meetings

5.3 GUI

5.4 Data model

5.5 Implementation

5.5.1 Code examples

5.5.2 Design Patterns & Principles

5.5.3 Unit test

5.6 Multi-threading and Concurrency

5.7 Sprint Review

5.8 Sprint Retrospective

6 Conclusion

References

[1] arla.com. *Meet Sidsel*. URL: https://www.arla.com/company/job-and-career/inside-arla/production/?id=43028 (visited on 05/20/2021).

[2] productplan.com. *What are stakeholders?* URL: https://www.productplan.com/glossary/stakeholder/ (visited on 05/22/2021).

[3] scrum.org. *SCRUM framework*. URL: https://scrumorg-website-prod.s3.amazonaws.com/drupal/2021-01/Scrumorg-Scrum-Framework-tabloid.pdf (visited on 05/23/2021).

[4] scrum.org. *What is SCRUM?* URL: https://www.scrum.org/resources/what-is-scrum (visited on 05/23/2021).

[5] sourcemaking.com. *Facade Design Pattern*. URL: https://sourcemaking.com/design_patterns/facade (visited on 05/20/2021).

[6] sourcemaking.com. *Factory Method Design Pattern*. URL: https://sourcemaking.com/design_patterns/factory_method (visited on 05/20/2021).

[7] tutorialspoint.com. *MVC Framework - Introduction*. URL: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm#:~:text=The%20Model%2DView%2DController%20(,development%20aspects%20of%20an%20application. (visited on 05/18/2021).

Appendices

A Appendix A: Introduction

A.1 Persona: Oscar

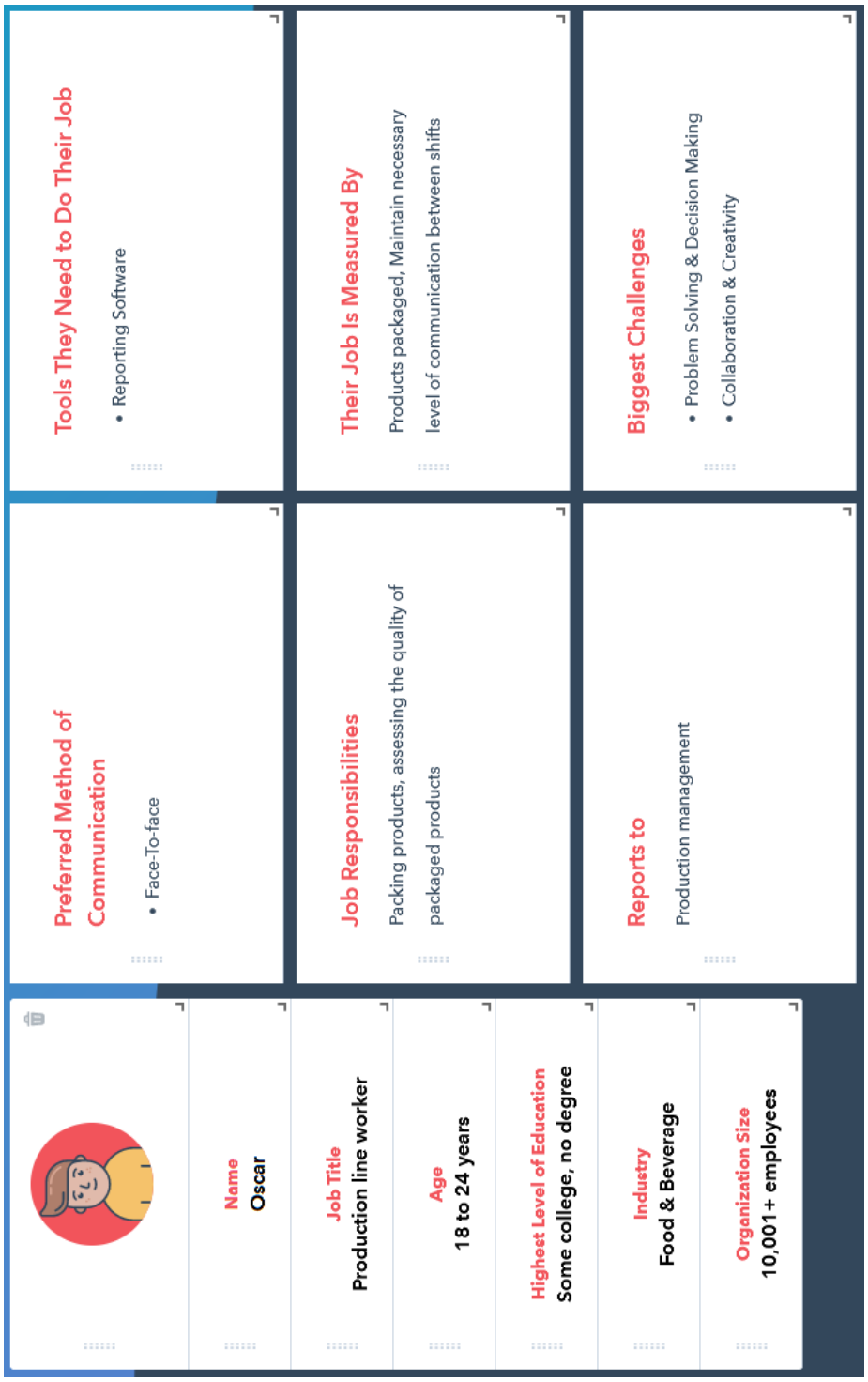

Figure 9: Persona Oscar