GITHUB:

https://github.com/AnthonyDTU/CovidMap

COVID-19 Map

Kursus: 62514

Object Oriented Programming

Semester Project

Anton Lage

s191270

11 maj 2021

Content

1 Introduction

We have been tasked to create a dashboard, summing up the COVID-19 situation in Denmark. A lot of these has popped up over the last 13 months, and now its our turn to participate. It is a fantastically relevant assignment, since it is undeniable that COVID *is* our new everyday life, and you might as well dive head first into the few opportunities presented by that fact.

I have chosen to do this project solo, since my usual group members either are not following this course, or dropped it due to personal matters. That being said, I threw my everything into this (within the limits of 4 other courses, a side job, sports, a girl and a social life), but I am happy with the outcome, and hope that the project lives up to expectations.

So, without further ado, I present you with my take, on a COVID-19 map.

2 Requirement Specification

2.1 Feature Requirements

The program **must have** the following features:

1. A visual representation of the corona situation in Denmark.
    - A map of Denmark, and its regions, where there is some indication of the COVID situation in the different parts of the country.
    - Data visualisation in the form of charts.
2. A way for the user to filter the data, and only be presented with what is relevant for that specific user.
    - The map should be intractable, so the user is able to select between the different regions, and only get KPI values from that region.
    - The possibility to filter the charts to a certain time period, and a specific municipal.
3. Data ranging back 1 month.
    - Data comes from SSI, and we will see what is available here.

When all of this is implemented it would be **nice to have** the following feature also:

1. An actual updater, where the program gets the data from a web-server, so that it is always updated, and not just a mock program with static data.
2. Data ranging back to the beginning of data collection, regarding COVID-19.
3. An export feature, where the user can take data out. Again this needs a sort feature, so only the data the user is interested in is exported.

2.2 Technical Requirements

The program is to be written in Java, using JavaFX and openjdk-16. The IDE used is going to be IntelliJ IDEA 2021.1. The target for the program is Windows 10, but other operating system should also theoretically support it, but this cannot be tested.

The program should be easy to use, and have a user friendly UI. Aesthetics are of course subjective, but a simple, beautiful, flowing UI is important, and I will do my best to compose everything into a great UI.[1]

The program should be easy to maintain and highly modular. This in terms means low coupling, where the different modules is as little depended on each other as possible. It also means high cohesion, where related modules and functions are closely located, and easy to find.

The program should be written with error prevention in mind, and throw exceptions where this is not possible to always check.

The project should be tracked in a git repository for version history, and incremental work progress.

I should also include a lot about performance requirements and memory limitations, but I do not know enough about what to expect to even make a guess here. What I do know is, that this is not that big of a project, and that it would be cheaper to upgrade the RAM with 16GB to be safe, than for me to find out how much is required, so it is not really relevant, but it should be included in a *real* requirement specification

2.3 Use case

The general use case of a COVID-19 map, including this one, is to give the user a quick and simple overview of the COVID situation. In this case it is only going to be relevant for people living in, or are otherwise interested in, Denmark.

[1] I did a much larger program, with much much more user interaction previously, were this approach of user-friendly-ness was make or break. It ended up being a make

3 Design

3.1 Overview

The program is written in with the concept of MVC in mind. In this implementation, the Controller knows about both the View and the Model, but neither of those know anything about the other two's existence.

Besides the Model, the Controller and the View, I wrote some custom classes and components, to help split up the project into more decentralized parts. These include the following:

- **Class** - DataFile
- **Enum** - ChartConfigurations
- **Class** - MapView
- **Class** - DataView
- **Class** - KPIField
- **Class** - DataChart
- **Enum** - RegionCoordinates

The MapView and DataView have their own initializer classes, both called by the object itself, if so chosen:

- **Class** - DataViewInitializer
- **Class** - MapViewInitializer

These intitializers configures all of the components to fit my application. Initializers for other applications could be written, if other use cases for the same component arose, but I only need the one, and this way, all configuration is located in the same class, which is then disposed of, and never used again.

The initializers servers as a kind of semi-controller for their respective components, until passed onto the actual controller and further passed onto the View, upon completion of the initialization.

3.2 UML Class Diagram

The UML Class Diagram can be found in appendix A

4 Implementation

4.1 DashboardView

The View has a very simple task. It's only job is to compose a view, for the scene to present. The view is made up entirely by a MapView and a DataView, placed in a grid. These two components can the be fetched by the controller, and since the MapView contains all the relevant controls for that, and likewise for the DataView, the controller can fetch everything it needs in one function call. This also means that everything is set with only two functions calls. One for the MapView, and one for the DataView.

4.2 DashboardModel

The Models job is to hold all the data loaded from the files. This is done in a DataFile object. One for each file loaded. Loading of the files is embedded into the DataFile, but it is all stored in the model. Since there is no real data manipulation, the model does not have any other jobs than storing and providing data.

4.3 DashboardController

The Controller has the responsibility for everything that happens. It does not do much itself, but it delegates a lot of work to the helper classes and components. Especially during initialization, where it tells the DataFiles to load and stores them in the model, makes sure the MapView and DataView initializes them selves, and passes it onto the view, as well as making sure that Buttons and ComboBoxes all add their respective event handlers.

4.4 MapView

As mentioned in the Design section, the core MVC utilize some helper classes, to help realize the program. One of these is the MapView, which makes up about half of the user interface. It consists of all the necessary components (Buttons, ImageView, Labels, etc) to make up that part of the UI. This helps group related components together, and improves cohesion.

4.5 MapViewInitializer

To configure the MapView, the MapView itself calls the MapViewInitializer, which is responsible for configuring all the components in the MapView. This includes setting location, size and color of the buttons, populating the ImageView, configuring the KPIFields, loading the initial data, and stitching it all together.

The class has one public function, which returns the initialized MapView. Inside of this, a number of private functions are called, each configuring one specific part of the MapView. The order they are called is not random, since some parts is needed for later configuration

(the buttons needs to be initialized, before they can be added to the Pane they are going to be contained in f.x.).

4.6 DataView

The DataView is very similar to the MapView. It makes up the other half of the UI, and, as well as the MapView, also contains all the components and configurations needed to realize it. It too, is also only composed of related components, for better cohesion.

4.7 DataViewInitializer

The DataViewInitializer performs the same tasks for the DataView, as the MapViewInitializer performs for the MapView. It initializes a DataView, in the configuration I need for this application, with ComboBoxes, Charts and ScrollView in the correct layout, as well as populating the charts with data.

4.8 DataFile

In the process of figuring out the data structure of this program, I ended up with this solution. Each file that is loaded, is stored in a DataFile object. This contains a map of the data, a list of line keys, and a list of data field keys. The latter two can then be used to look up specific data cells.

The system works exactly like you would read the file yourself, if one was tasked with reading the content from a specific line, in a specific data field. The object has an empty constructor (i.e. creating an empty file), and then a function can be called, which loads a file into the data fields described earlier.

The model then holds a DataFile object for each actual file that is loaded, which can be accessed and read from. When accessing data, the program fetches a whole DataFile, since it contains both keys and data, and holds all necessary information for accessing data.

4.9 DataChart

This is not an actual chart, since the object does not extend the javafx.charts. It does however contain four different chart initializer functions, which is more manageable than having them spread out over the rest of the classes. Three of the functions is used when initializing the DataView, and initializing the charts, and the last function is used when updating the charts.

4.10 KPIField

The KPIField is there for a more manageable object, than several individual labels. The object contains a title and value label, in the correct layout for this application. It also contains

functions for updating and reading the value label. There are four of these contained in the MapView, located in a GridView just above the actual map, and the values updates whenever the user selects a different region to show data from.

4.11 ChartConfigurations

The ChartConfigurations enum contains configuration information about how the chat related to every file should be compiled. This includes stuff like title, legend titles, fetching of the correct DataFile, and stuff like which lines to exclude from the respective files.

This massively simplifies the configuration of the charts, since every deviance from the most basic procedure can be accounted for here.

4.12 RegionCoordinates

This enum contains coordinates for where to place the buttons on the map, and it contains functions for getting each of them.

5 Test

5.1 JUnit Tests

I do not see a huge benefit for unit testing on this particular application. The reason I do not think unit testing is very useful here, is because it is mostly a visual program, and there is not a whole lot of functionality and logic. Even when there is, the only thing done, is visual updates, which is hard to write unit tests for.

However, I have made a few, just to add more completeness to the project, and demonstrate, that I know how to use the framework for basic testes.

- **DashboardControllerTest** - This tests the creation of the model, the view and the controller, as well as ensuring that files are loaded, without them being null.
- **DataViewInitializerTest** - This initializes a DataView, and checks that the correct amount of components and keys are created, as well as the content in the filter ComboBoxes being correct.
- **MapViewInitializerTest** - This initializers a MapView, and checks that the correct amount of components and keys are created, as well as the fact that there is a map loaded.
- **KPIFieldTest** - This initializes a KPI Field, and checks that the correct amount of components are created, and tests that the updated value function works properly.

The unit tests are located in the Tests folder in the solution.

5.2 Accept Test

The application do not do a whole lot, and does not have a lot of functionality or user input. This limits the amount of test conceivable, and the acceptance test is going to be quite manageable.

5.2.1 Test 1 - Loading Files

Make sure that all files are loaded correctly. If the program does not find anything, it will throw an exception on startup, and a message will be shown to the user, but it is also necessary to check that files loaded is loaded correctly.

This is done in debug mode of the program, reading that the different files has the correct number of lines and data fields, as well as there being data in the fields.

Results: All files observed to be loading and storing correctly.

Passed

5.2.2 Test 2 - Map interactivity

Test insuring the interactivity of the map should be performed. The functionality of switching between all five regions, and the entire nation should be tested, in compliance with the following procedure:

Danmark → Nordjylland → Midtjylland → Syddanmark → Sjælland → Hovedstaden → Danmark → Nordjylland → Danmark → Midtjylland → Danmark → Syddanmark → Danmark → Sjælland → Danmark → Hovedstaden.

When each of the region buttons, or the map itself for Denmark, is pressed, the KPI Title should update to the region, and the value labels should update with the values contained in *Region_summary.csv*, in the related region/KPI.

Results:
The map behaves as expected. The KPI title and values updates according to the values they should, and no errors has been encountered.

Passed

5.2.3 Test 2 - Time Filtering

Test insuring the ability to filter the charts by a certain time period should be performed. The avaliable time periods is limited to

- All Time
- 30 Dage
- 7 Dage

When these are selected, the following charts should immediately update, and show only the specified time period:

- Positive Cases
- Tested
- Admitted
- Deaths

The pie charts containing data about cases by age and cases by sex should **not** be updated at all, since there is no data other than a total.

Results:
When selecting a new time period, the charts behave as expected, and desired. They update right away, and correctly displays the desired time period.

Passed

5.2.4 Test 4 - Municipality Filtering

Test insuring the functionality of the municipality filter should be performed. As well as working on its own, this should also work in conjunction with the time filtering functionality. Since there is so many municipalities in Denmark, selecting a few will do. To make sure it also works with the time period selection, the test pattern should be as following, with the starting configuration of *All Time*, *Danmark*:

Aarhus $\rightarrow$ **30 Days** $\rightarrow$ **Skanderborg** $\rightarrow$ **7 Days** $\rightarrow$ **Herning** $\rightarrow$ **30 Days** $\rightarrow$ **Syddjurs** $\rightarrow$ **All Time** $\rightarrow$ **Danmark**

When updating the municipality, only *Positive* and *Tested* charts should reflect the data of the municipality, since there is no data of this in relation to admitted or deaths. These will just continue to show data from the whole nation, but will of course still reflect time filtering.

Results:
When running through the sequence specified, everything acts as expected. *Positive* and *Tested* change both according to time and municipality filtering, and *Admitted* and *Deaths* changing according to time filtering. No problems has been encountered.

Passed

6 Conclusion

The application has reached it final state, before being handed in. More could definitely be done, and I must admit, I had hoped to add a few more features than I have, especially fetching the data from the network, so that it is up to date at any time.

But that being said, I am happy with the way the features I did implement, turned out.

The MapView is intractable, and the user can click around the different regions. The buttons for each region changes size, depending on how bad the COVID situation is, for indication without having to read the actual values.

Data is also displayed nicely in graphs, which can be filtered by time and municipality, and everything in that feature set works just fine, and is operated in a user friendly fashion.

So all in all, I achieved the goal of creating a COVID Dashboard. It does what it is supposed to do, and it is not cluttered or incomprehensible in any way. I had a bit more trouble getting used to the JavaFX framework than expected, which took time away from a broader feature set, but that is how things go.

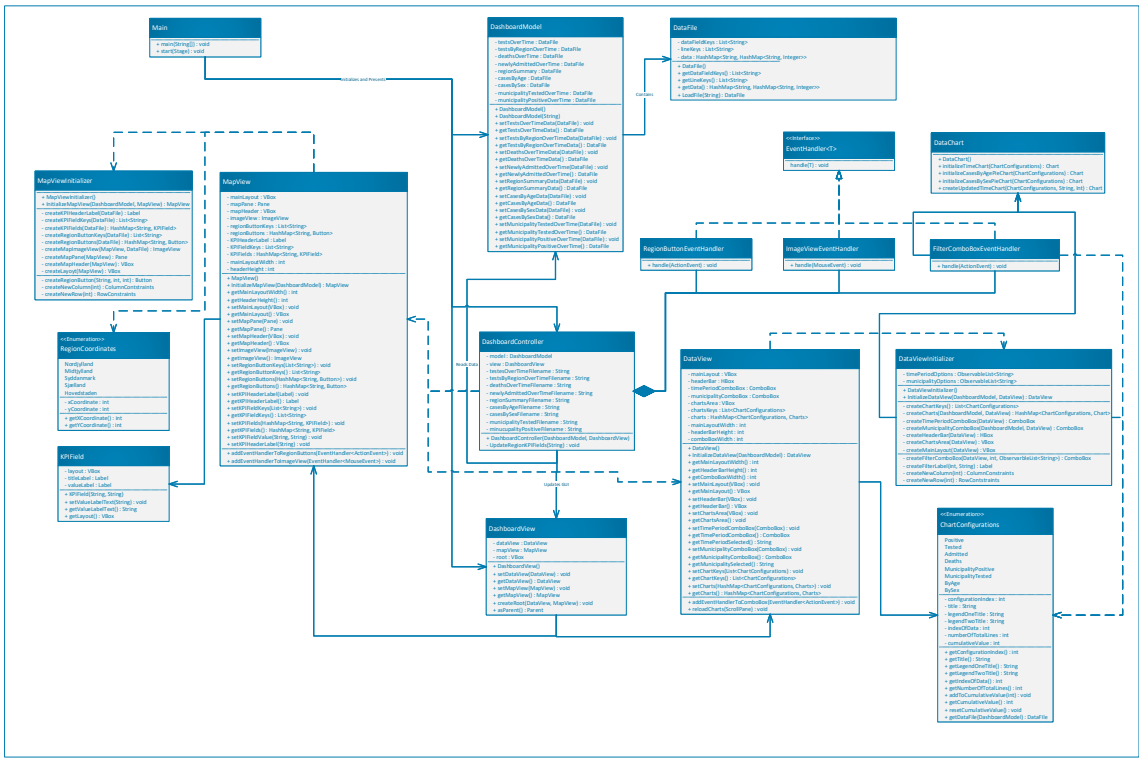
And lets then end with a bit of reflection. I have really enjoyed this project. It was great to have something to really dive into, and to have so much creative freedom to dream up a vision of how I would like the final product to turn out.

Now there is only one thing left to do, which is presenting the project to you. I will be looking forward to that, as I hope you are as well.

7 Appendix

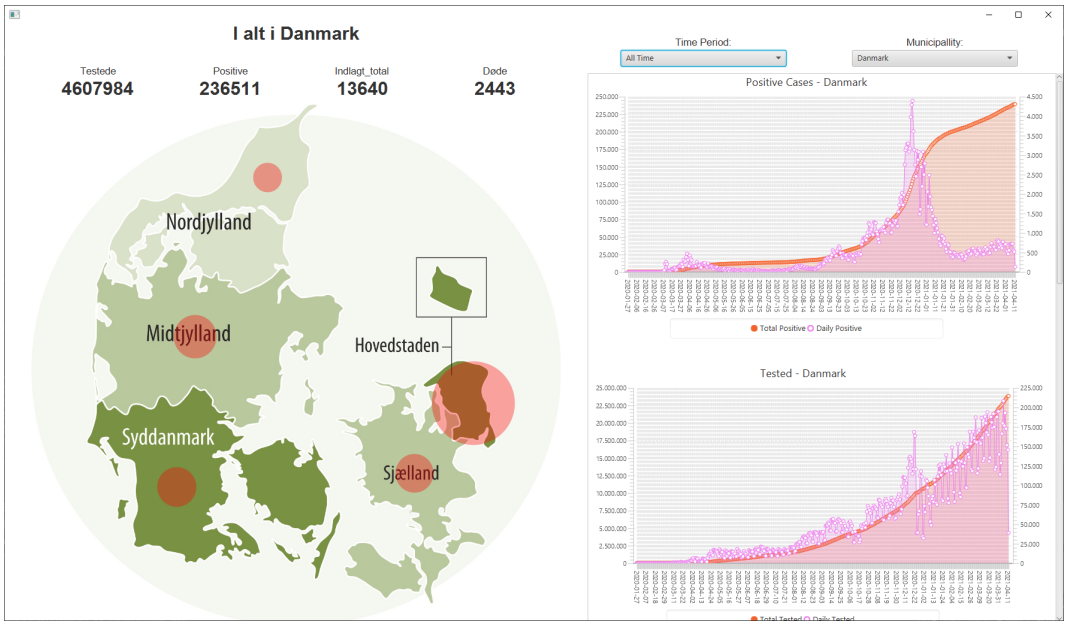
A UML Class Diagram

Not very decipherable, but it is also included as a separate PDF file *UML Class Diagram.pdf*

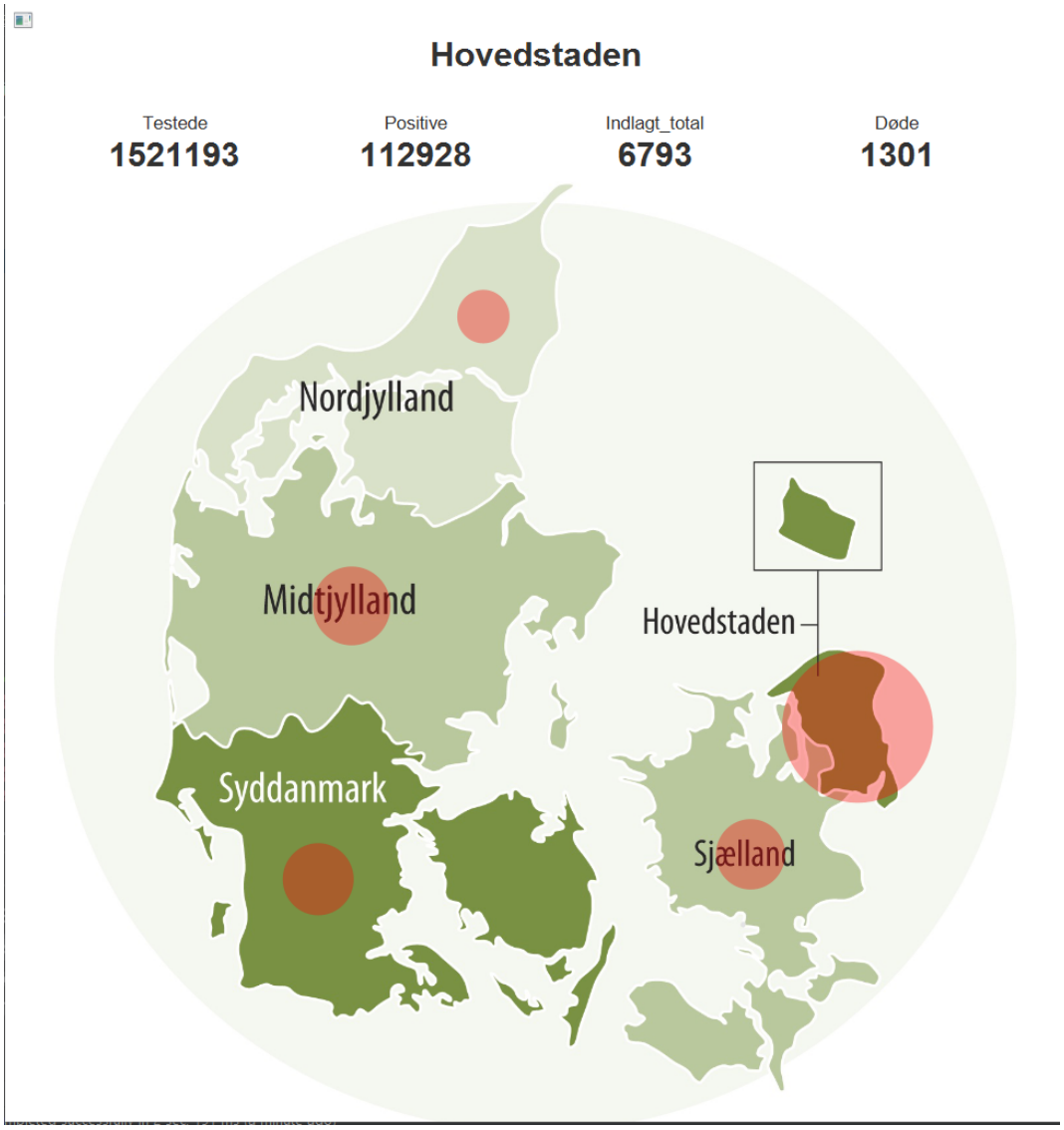


B Screenshots

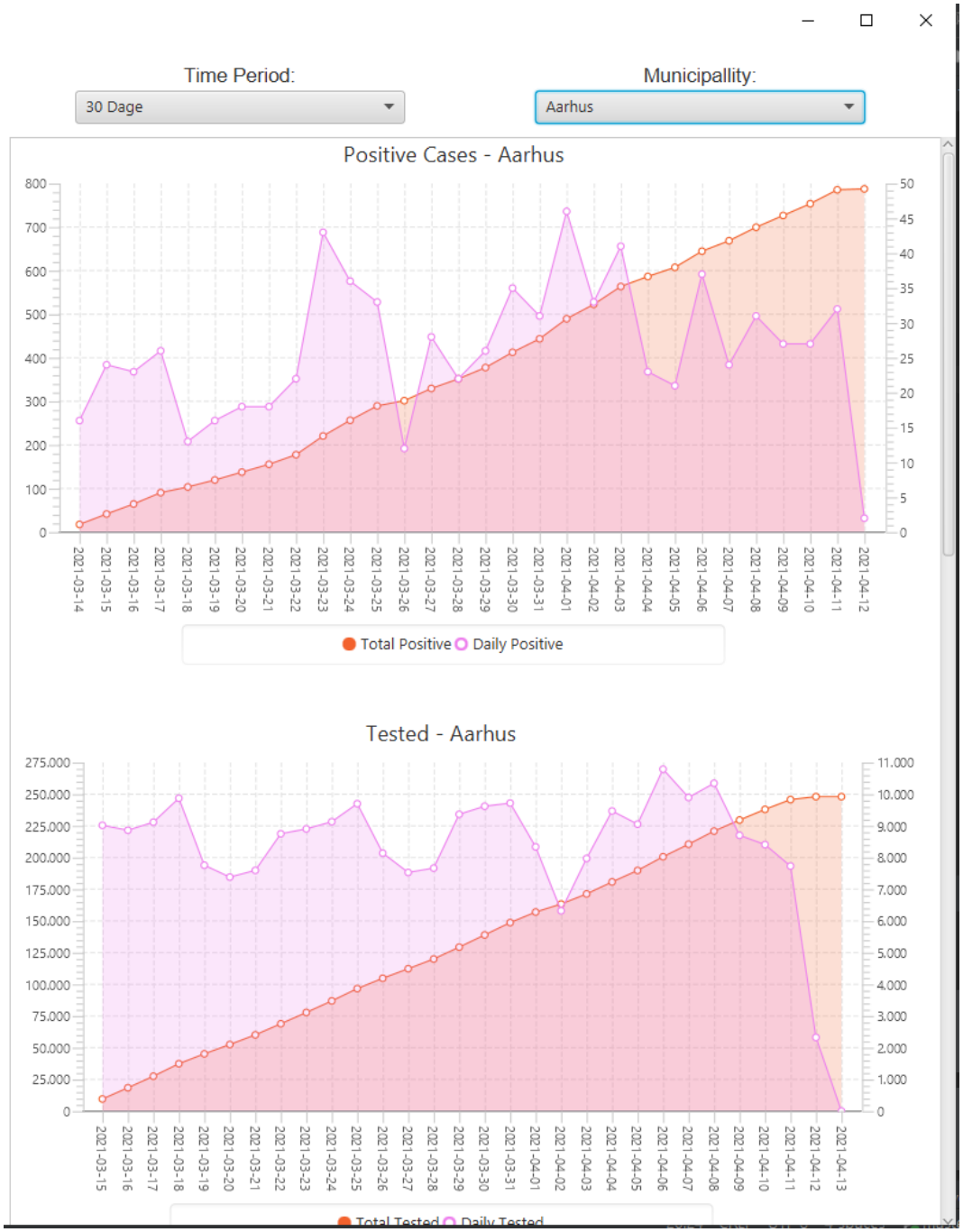
B.1 Complete view of the UI from starting configuration



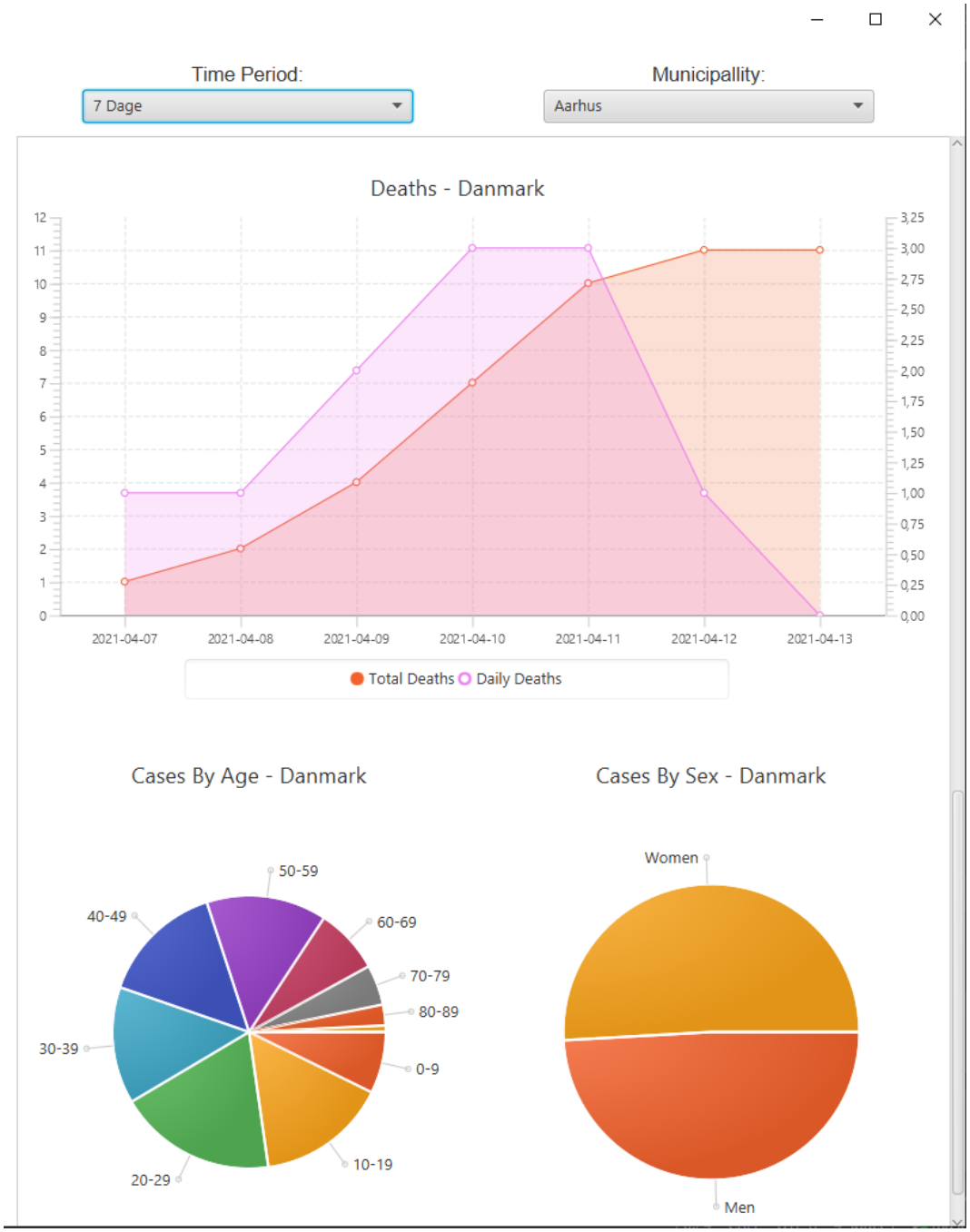
B.2 Detailed view of the MapView after selecting *Hovedstaden*



B.3 Detailed view of the DataView after filtering for 30 Days in Aarhus



B.4 Scrolled down view of the DataView, to show piecharts



C Resources

Referenser

[1] MultiTypeChart, used as a package for formatting charts. The benefit gained from using this was in particular the two y axis for each chart.
`https://github.com/KilianB/JavaFXMultiChart`

[2] Tutorial used for charts in javaFX. Links going to more specific charts also found here, and some of these have also been used.
`https://www.tutorialspoint.com/javafx/javafx_charts.htm`

[3] Toturial for MVC JavaFX project, without FXML.
`https://www.youtube.com/watch?v=dTVVa2gfht8ab_channel=DerekBanas`

[4] I have drawn most of my inspiration from the John Hopkins University COVID map, which can be found here
`https://coronavirus.jhu.edu/map.html`

[5] ComboBox introduction
`https://docs.oracle.com/javafx/2/ui_controls/combo-box.htm`