MVVM Exercises

# Exercise 1 – Run slides example

Copy the code from slides 75 to 86. Follow the package structure shown in slide 87.

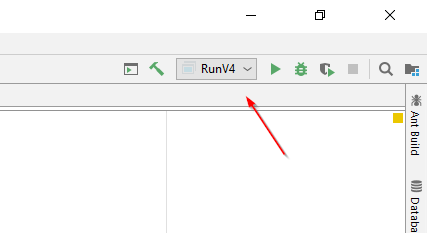
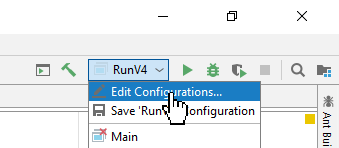
Run the program.

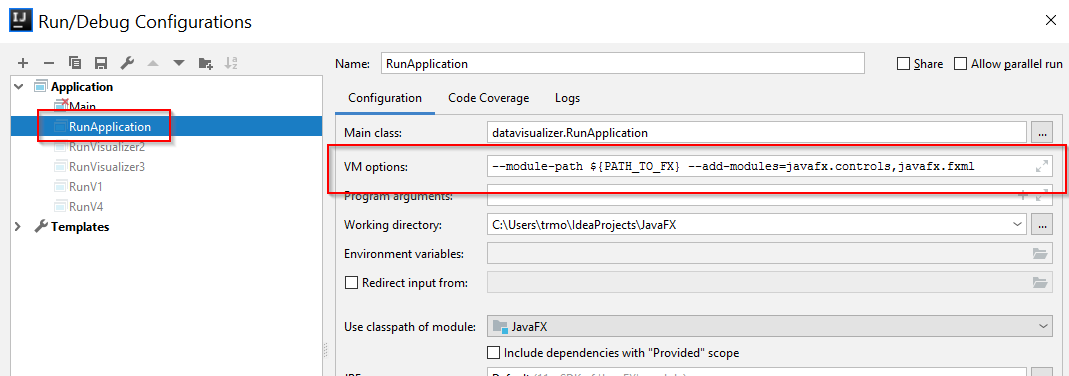
You may get an exception, when running this, if you forgot to set VM parameters.

There are two fixes: Set VM parameters, or move the main method

Fix 1

In IntelliJ it’s done here:



Select your class with the main method on the left. Add VM options on the right.

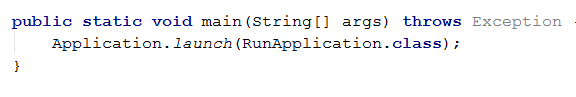
It’s described in more detail here, for both Eclipse and IntelliJ:

<https://openjfx.io/openjfx-docs/#IDE-Intellij>

Step 4, add VM options.

Fix 2

You should be able to move this piece of the code in RunApplication:



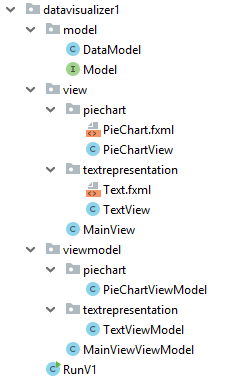
To a separate class, and just run this main method instead.

Run the program, click the Update button once in a while. Verify it works as expected.

# Exercise 2

We’ll expand on the previous exercise. This time adding a new View, which can modify the data.

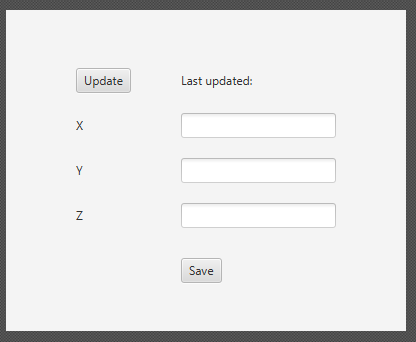
Expand your package structure to look something like this:



You’ll notice two new packages call textrepresentation. These will contain the new classes.

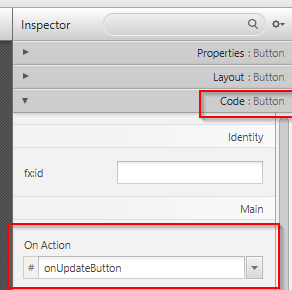
UI

Design an fxml to look similar to this:



I have used the GridPane as the outer container.

The Button has the text Update, and the On Action like this:

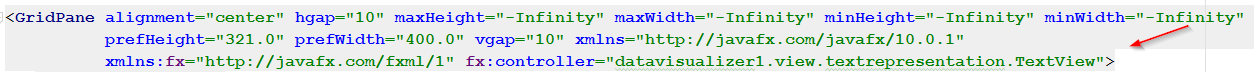


When saving, you will see this in the fxml file:

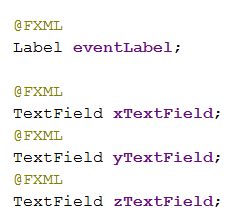


There must then be a method with the identical name: onUpdateButton in the TextView class.

Remember to specify the “Java controller” at the beginning of the xml:



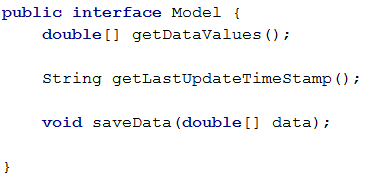
The x, y, z are just labels. The TextFields have fx:id, similar to the Button, shown above. We can reference the TextFields using the fx:id from the code:



The Save button must also have a On Action name in the fxml, so there is a corresponding method in the View class.

Model

When you have designed the UI, modify the Model interface, so it has a method to store data:

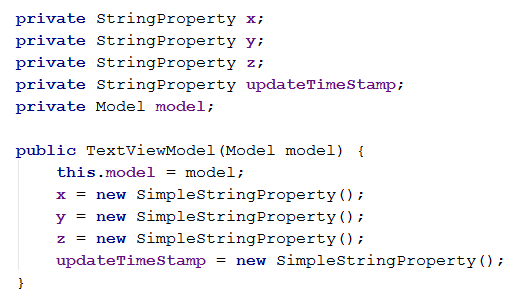


Implement the saveData method. It just takes the array, and puts the values into x, y, and z.

ViewModel

Next, do the ViewModel. It will look very similar to the PieChartViewModel.

There should be a StringProperty private field for x, y, z. The data may be doubles in the Model, but the GUI must present it as Strings, and so it is the responsibility of the ViewModel to convert:



There should be a method, updateTextFields, which will get data from the Model, and insert into the three StringProperties.

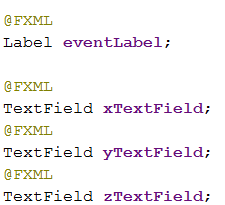
Also, here, get the latest timestamp from the Model, and update the updateTimeStamp StringProperty.

Create a get method for each of the StringProperties, so the View can bind to them.

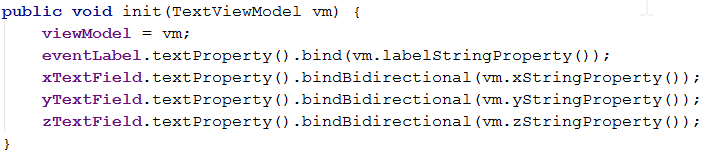
Create a method, call it saveChanges. It should take the values from the x, y, and z properties, convert them to doubles, and then create a double array with the values. Call the saveData method on the model with this array.

View

Your view should have the fxml component references:



All our views must have an init method. Because JavaFX automatically creates our View classes (JavaFX Controllers), we cannot give parameters to our constructor. Therefore, we need an init method:



Here, you bind the View TextField properties to the StringProperties from the ViewModel.

In the two methods, corresponding to the two buttons, call the relevant methods on the ViewModel.

MainViewViewModel

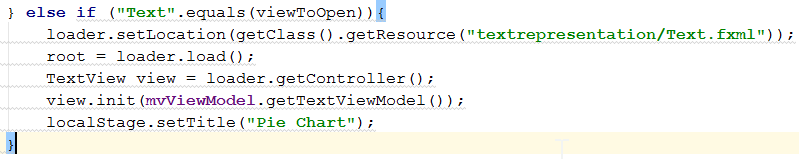
You’ll need to add a bit to this class too. Remember, the MainViewViewModel is just responsible for creating all other ViewModels. Instantiate a TextViewModel, store it in a field variable, and create a get method.

MainView

The MainView is responsible for creating, showing, and changing view.

You’ll need to change the existing openView method.

Add the following:

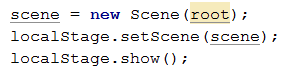


Some of the above variables were defined above the if statement.

Now, if you only whish to show one view at a time, you’ll take your stage and set the view, and then show the view.

If we want to show multiple views simultaneously, you’ll have to create multiple stages:





Finally, you’ll need to call the openView method twice, once with “Text” parameter, once with “PieChart”, to open both views.

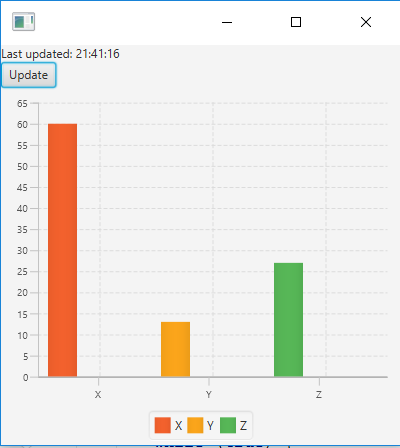
Run the program

Now, run the program. You should see both views: the text editor, and the pie chart. The data in the Model should be update regularly because of the thread started in the main method.

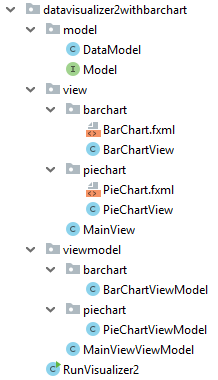
You can click the update buttons to show the latest. You can edit data in the text fields, save it, and update the pie chart view to see it reflected here.

# Exercise – Bar Chart

We’ll be adding a new View, to show the data in a different way.



Add packages to your structure, similar to:



You’ll be expanding on the previous exercise, so you should also have packages with the text view.

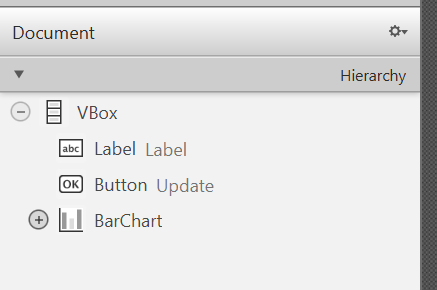
We have added to packages called barchart, one with BarChartViewModel class, the other with BarChartView and BarChart.fxml

Fxml

The fxml is created in the SceneBuilder:

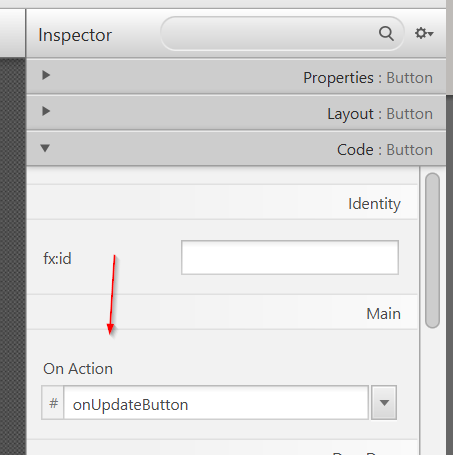


I have used a Vbox, with a Label, a Button, and a BarChart.

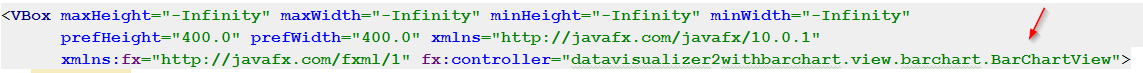


They have fx:id so they can be referenced from the code in the View.

The Update button has an On Action method name, so a method will be called when clicked

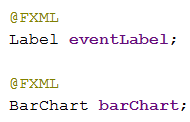


Update the fxml to say which Controller is used:



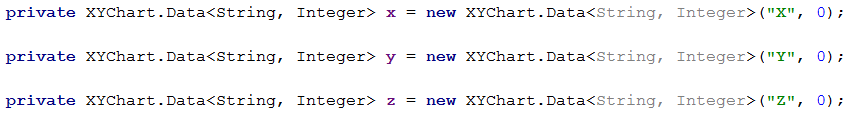
View

In the View class, create corresponding field variables:

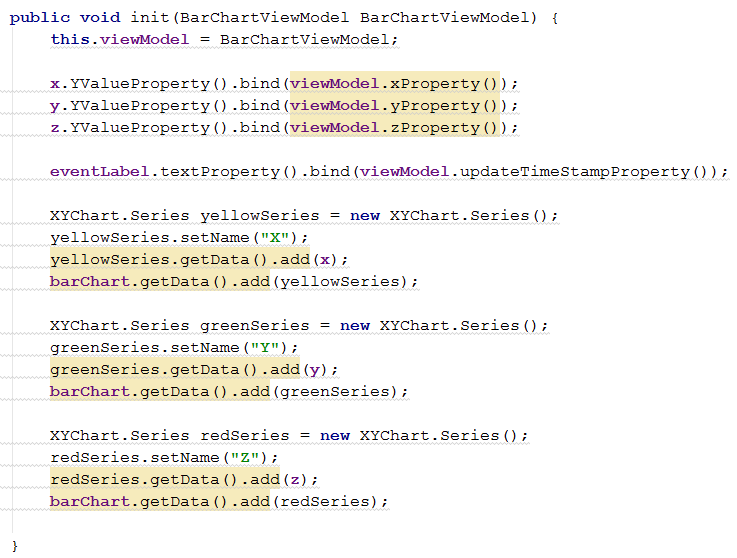


Create the update method. This will be similar to the method in PieChartView. Leave it empty for now, you don’t have the ViewModel.

You’ll need the following field variables:



In the init method, put the following:



You can read more about the barchart here (along with most other java fx components)

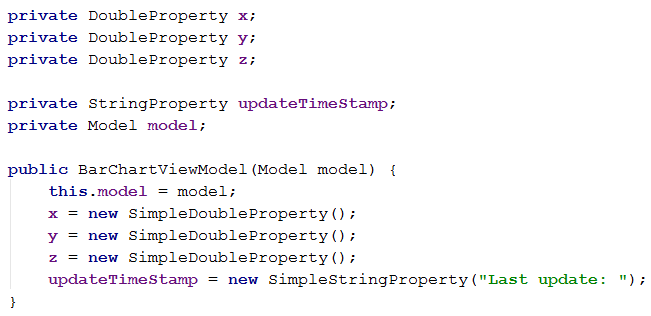
http://tutorials.jenkov.com/javafx/barchart.html

In short, each data series has a different color. We want each value of the model to be a column of different color. So we create three series, each series has a single data.

You don’t have the ViewModel yet, so some of the lines here will complain. So let’s fix this

ViewModel

In the ViewModel, you’ll again need a method to fetch the latest values from the Model (also get the latest time stamp), and update private field Properties:



Create a get method for x, y, and z property, so that the View can access them and bind to them.

MainViewViewModel

In this one, like before, create and store a BarChartViewModel, with a get method.

MainView

Add an extra “else if” statement in openView method, where you instantiate the BarChartView.

Run the program

Run the program, verify that you see all three views: PieChart, Text, BarChart.

Verify that you can edit the values in Text view, and update the others to see the changes.

# **Extra** Exercise – StackedBarChart

Create another view, this time using the StackedBarChart. Follow similar steps as in the previous two exercises.

# **Extra** Exercise – Observer pattern

You’re going to implement the observer pattern.

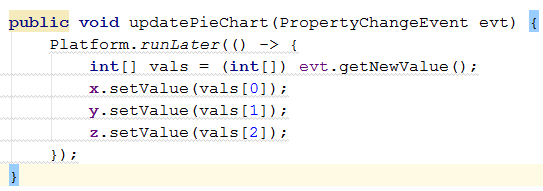
Let the Model be an Observable, add appropriate methods to the interface to add a PropertyChangeListener. Let the DataModel have a PropertyChangeSupport.

When new data is calculated, fire an event.

When a new timestamp is created, fire an event.

In the ViewModels, listen for these events, and update the properties. This should then be reflected in the view, and you should see the charts being updated automatically. There is something tricky here, because we have a JavaFX thread running, but we have also created a separate thread, which updates the model. When the model is updated, an event is fired, this is “caught” in the ViewModel, which then updates a property. Now, only JavaFX threads are allowed to modify properties, and the other thread we started is not a JavaFX thread.

So you’ll have to wrap the update to properties like this:

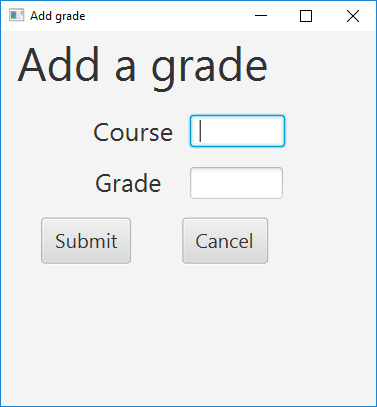
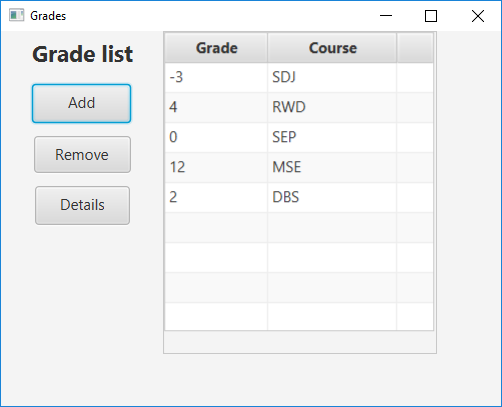


# Exercise – Grade list, A

Download GradeListApplication.zip and copy it into your project. You may need to verify the fxml files points to the correct Controllers.

Run the main method in class Main.

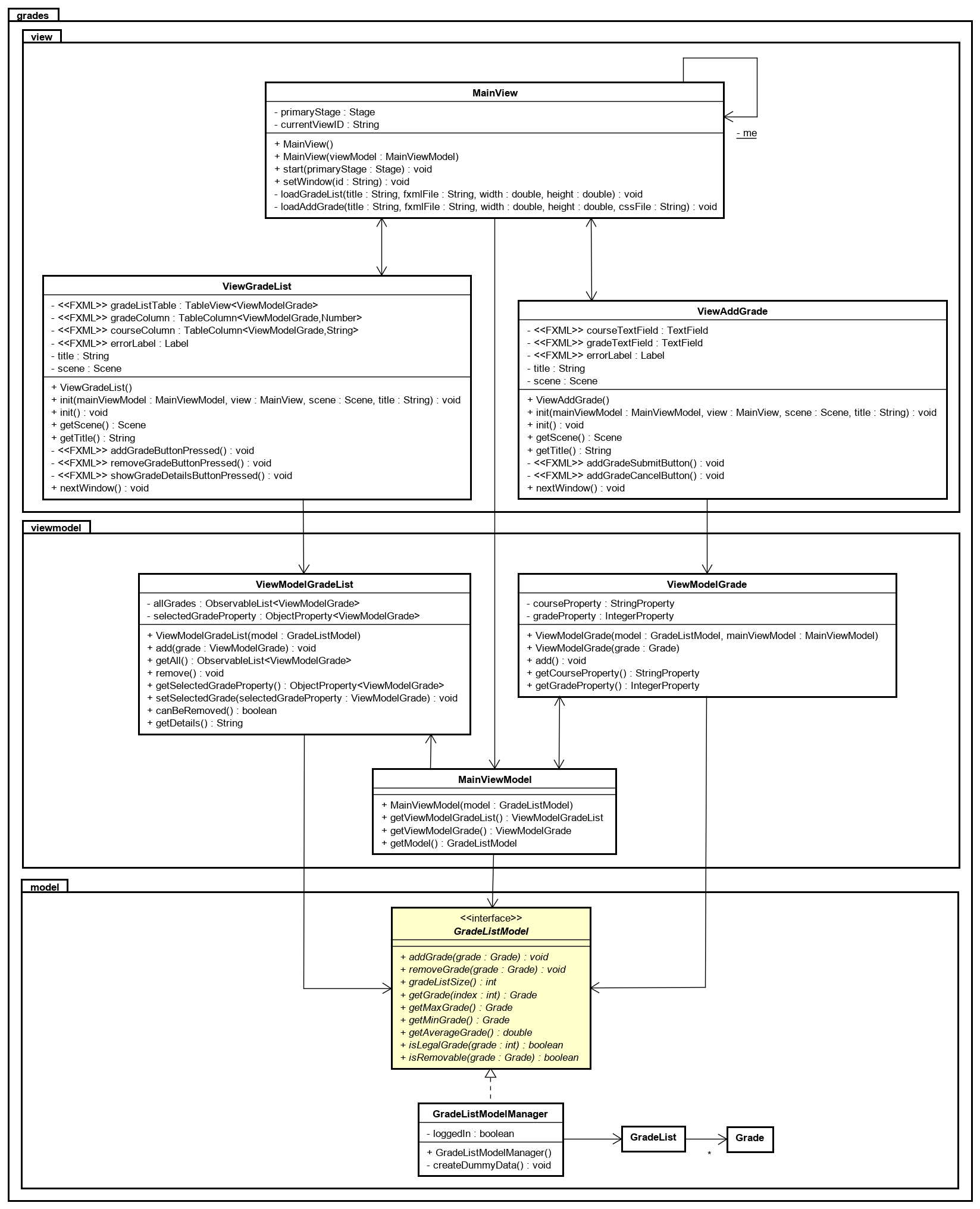
**After running main the output should be like the following (values for grades could be different):**



Test it, remove grades, add grades and see details

# Exercise – Grade list, B

The Exercise is to convert the already existing application from the previous exercise (given in GradeListApplication.zip) to an MVVM (Model-View-ViewModel) version exactly as shown in the class diagram below



You have to do the following:

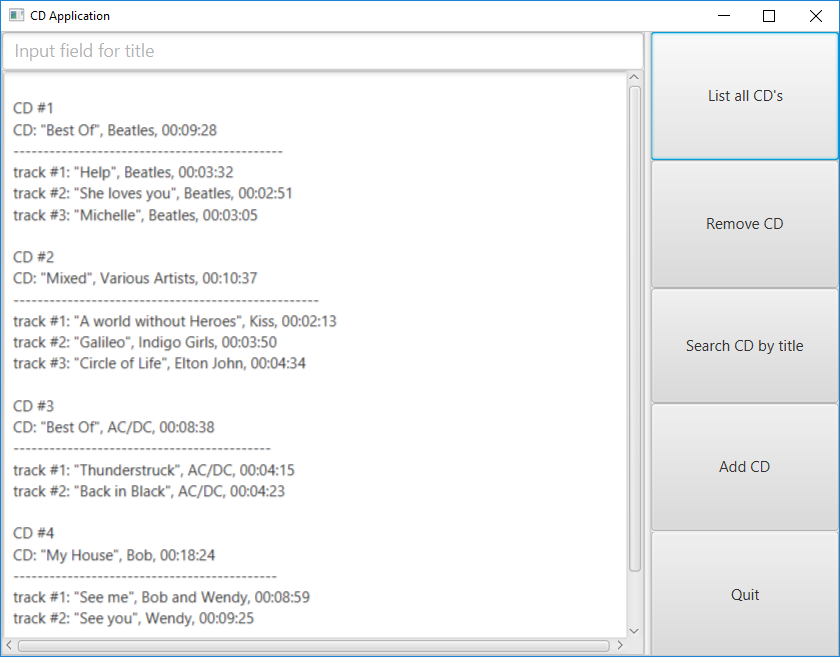
* Make sure classes are in the packages shown (move ViewModelGrade into a package grades.viewmodel)
* Create the two ViewModel classes ViewModelGradeList and ViewModel grade
  + Instance variables and methods exactly as in the class diagram (3 and 4 instance variables, respectively)
  + Constructors (in ViewModelGradeList you also add all grades from Model into the ObservableList instance variable (in the present version this was in the view class)
  + In method add in ViewModelGrade make sure you call add in ViewModelGradeList (to update the ObservableList and hereby the table)
  + Try to see if you can figure out how to implement each of the methods
* Update ViewGrade and ViewAddGrade with the following
  + Remove any connection to model (instance variable and references)
  + Add the ViewModel as instance variable and pass it to methods init
  + Update method init to include
    - In ViewAddGrade, bind bidirectional the two InputFields to the ViewModel properties
    - Try to figure out what to include and what to delete
  + Try to implement the methods such that you are convienced that the responsibility for the view is solely to present what will alredy be done in the ViewModel.
* Create a class MainViewModel
  + Instance variable for both ViewModel classes and the Model
  + A constructor creating the two ViewModel objects
  + Getters for Model and the two ViewModel objects
* Change the Main method according to your changes of design and implementation

Run the application.

# Exercise – CD list, A

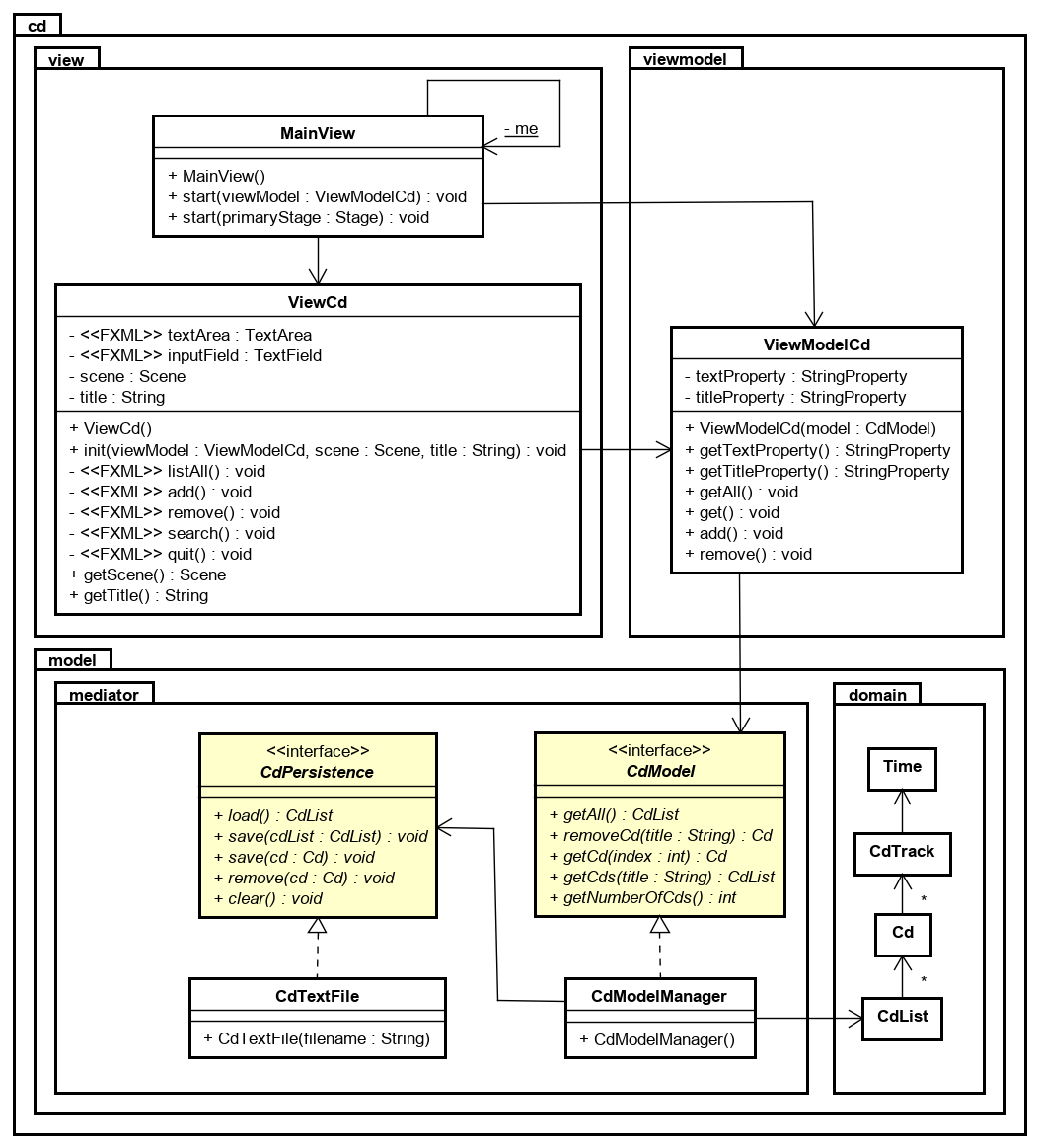
Download CDApplication.zip including source files for the CD application from the slides. Import this into Eclipse/IntelliJ. Run the main method in class Main.

**After running main and clicking the “List All CDs” button the output should be like the following:**



# Exercise – CD list, B

The Exercise is to convert the already existing application (given in CDApplication.zip) to an MVVM (Model-View-ViewModel) version exactly as shown in the class diagram below



You have to do the following:

* Packages
  + Move CdList, Cd, CdTrack and Time into package cd.model.domain
  + CdPersistence and CdTextFile in class cd.model.mediator
  + MainView, ViewCd and ViewCd.fxml in package cd.view
* Create interface CdModel (in cd.model.meditor package) with the following methods:

public CdList getAll();

public void addCd(Cd cd);

public Cd removeCd(String title);

public Cd getCd(int index);

public CdList getCds(String title);

public int getNumberOfCds();

* Create class CdModelManager (in cd.model.meditor package) The Model Manager keeps the Model’s state – in this case only a CdList instance variable. A second instance variable is of type CdPersistence.
  + The constructor is loading the model/CdList from file (and this part is then deleted from class Main). Make sure that your relative path to the text file is correct (e.g. "src/cd/model/mediator/cds.txt" and that your CdList instance variable is initialized
  + All methods are simply delegating the work to CdList and could each be implemented with a single statement.
* Create a class ViewModelCd (in a package cd.viewmodel)
  + A CdModel instance variable
  + Two StringProperty instance variables, one for title and one for output
  + A constructor taking the model as arguments (and initializing the CdModel instance variable). Both property instance variables are also created.
  + Getters for both property instance variables
  + A method getAll() setting the text property to a string version of the CDList you get from the model
  + A method get() getting the CD from model with the value stored in the title property as the title. If there is no value in the property then set a proper message in the text property otherwise set a string with the Cd's found. [copy the logic from ViewCd method search]
  + A method remove() with the logic coming from ViewCd method remove
  + A method add() simply setting the text property to "ADD: NOT IMPLEMENTED"
* Update ViewCd.fxml with fx:controller="cd.view.ViewCd" i.e. in the correct package
* Update ViewCd with the following
  + Remove any connection to model (CdList instance variable and references)
  + Add a ViewModelCD as instance variable and pass it to method init
  + Update method init to include
    - A statement to bind the InputField to the title property in the view model. This bind has to be bidirectional such that a change in the one is reflected in the other and the opposite (use method bindBidirectional)
    - A statement to bind the TextArea to the text property in the view model (use method bind as the TextArea is never directly changed)
  + Methods remove() and search() are now reduced to one-statement methods delegating to the ViewModel
* Change Main to the following

import cd.model.mediator.CdModel;

import cd.model.mediator.CdModelManager;

import cd.view.MainView;

import cd.viewmodel.ViewModelCd;

public class Main

{

public static void main(String args[])

{

CdModel model = new CdModelManager();

ViewModelCd viewModel = new ViewModelCd(model);

MainView view = new MainView();

view.start(viewModel);

}

}