Meeting Planner – Protocol

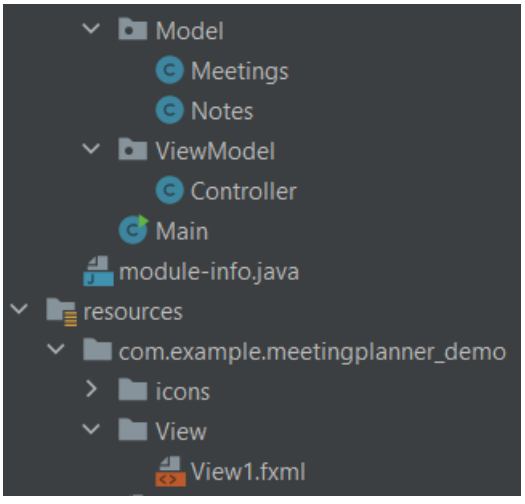
1. Application Architecture

A. MVVM pattern – UI

- I started the development of the project by creating the user interface via javaFX with the help of the SceneBuilder and some manual improvements in the .fxml file. After that I created the Controller class and appended it to the frontend .fxml file, as well the Meeting and Notes object classes.
  Then I structured these four components into three categories based on the MVVM – pattern:

  - **Model** (Meetings, Notes class)
  - **View** (javaFX .fxml frontend file)
  - **ViewModel** (Controller class)

- The **Model** section contains the two object classes used in this project – Meetings and Notes. These classes contain the basic implementation (constructors, getters, setters) and no additional methods.
- The **View** section contains the .fxml file which in turn contains the whole javaFX User Interface and is assigned a controller which handles the user input.
- The **ViewModel** section contains the controller class linked to the .fxml UI and contains methods for handling of user action in the UI (button clicks, selcting a meeting from a table) and passes the input from the UI to the businessLayer below.



1. MVVM pattern structure

B. Business Layer

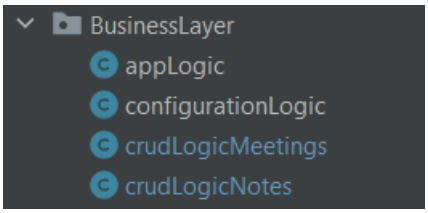
- Below the UI Layer I implemented the Business Layer, which recieves parameters from the UI layer above, handles them, possibly passes them to the Data Access

Layer below. The business layer can therefore communicate with both the DA and the UI – Layers as they are directly one layer below, respectively above it.

- The Business Layer is structured into four seperate parts:

  - **AppLogic** (Basic app functionalities)
  - **CRUDLogicMeetings** (CRUD validation for Meetings class)
  - **CRUDLogicNotes** (CRUD validation for Notes class)
  - **ConfigurationLogic** (Logic for config properties handling)

- The **AppLogic** section contains all basic functionality methods such as fetching complete lists of meetings/notes from the database or generating a PDF meeting report, these functions have to be called from the UI layer (Controller) and either return a list of results back or work with the passed parameters such as in the PDF generation.
  Here we can also find basic functions for minor validation (valid Integer/time...) or format parsing, which are used throughout the Business Layer.
  The selected meeting parameters are also stored here, whenever a user clicks on a meeting in the UI layer.
- The **CRUDLogicMeetings** section contains all related CRUD functionality methods for the Meetings object (Reading method is in AppLogic), as well as a validator for the received user input parameters from the UI layer.
- The **CRUDLogicNotes** section contains all related CRUD functionality methods for the Notes object (Reading method is in AppLogic), as well as a validator for the received user input parameters from the UI layer.
- The **ConfigurationLogic** section contains a single method, which takes a String parameter and returns an entry from the config file whose name corresponds to the parameter. It is used to fetch the config entries in the Business and Data Access layers.



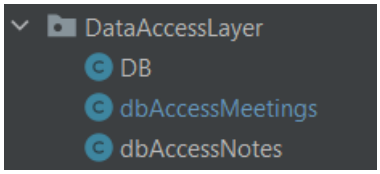
2. Business Layer structure

C. Data Access Layer

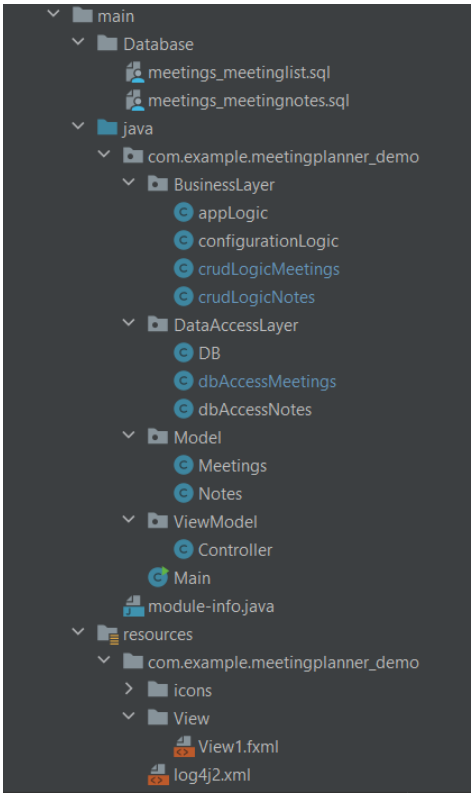
- The Data Access Layer is located below the Business Layer, which means it can only communicate with the Business Layer and not with the UI Layer directly. It contains the entire logic for interacting with the database (CRUD).
  The Data Access Layer is split into three parts:

- **DB** (Basic database interaction)
- **dbAccessMeetings** section (CRUD for meetings)
- **dbAccessNotes** section (CRUD for notes)

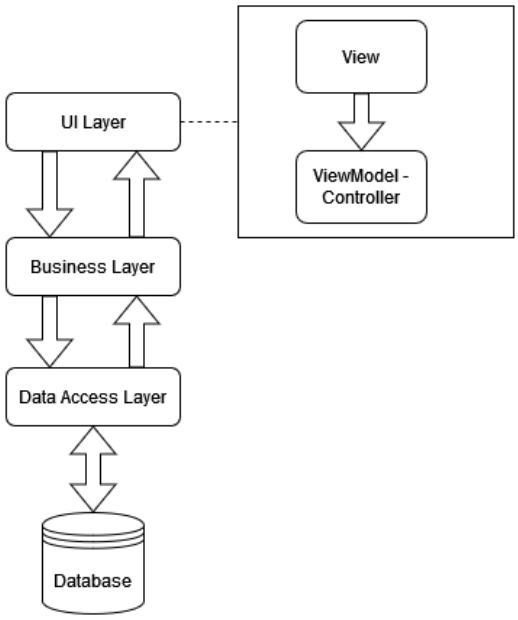
- The **DB** section contains the basic database interaction functionality for establishing a connection to the DB, executing a query, as well as a special update query execution that returns a Boolean depending on success or failure to find a corresponding object in the DB.
- The **dbAccessMeetings** section contains all CRUD methods for handling meetings in the database.
- The **dbAccessNotes** section contains all CRUD methods for handling notes in the database.



3. Data Access Layer structure



4.Complete project structure (without tests)



5.Project Layers Diagramm

2. Design Patterns

A. Builder Design Pattern

Motivation: The builder pattern is useful in constructing complex immutable objects in a step by step manner thus solving the telescoping constructors problem. In the project implementation, the builder pattern is used in the Meetings object class and provides an easier and more fluent way of instantiating new Meetings objects. It enables setting a mandatory parameters which need to be passed to the constructor at every new instantiation and also the optional ones, which can be appended to the constructor but do not have to be. This was espacially usefull for unit test purposes or other functionalities in which only a few attributes of the Meeting class are required to be set.

Implementation:

```
public static class meetingsBuilder {
    private Integer ID;
    private final String title;
    private String startDate;
    private String startTime;
    private String endDate;
    private String endTime;
    private String agenda;

    public meetingsBuilder(String title){
        if(title.isEmpty()){
            this.title = "DefaultTitle";
        } else
            this.title = title;
    }
    public meetingsBuilder ID(int ID){
        this.ID = ID;
        return this;
    }
    public meetingsBuilder startDate(String startDate){
        this.startDate = startDate;
        return this;
    }
```

A) The Builder constructor contains the parameters which are made mandatory in this case it is only the Meeting Title. The other properties each have their own set function which is optional to be called in the

```
public Meetings build(){
    return new Meetings( builder: this);
}
```

B) At the end of each new Meeting object instantiation needs to be the build() method call which in turn calls the Meeting constructor and passes the created Builder Object

```
public Meetings(meetingsBuilder builder){
    this.ID = builder.ID;
    this.title = builder.title;
    this.startDate = builder.startDate;
    this.startTime = builder.startTime;
    this.endDate = builder.endDate;
    this.endTime = builder.endTime;
    this.agenda = builder.agenda;
}
```

```
String modifier = "createMeeting";
Meetings newMeeting = new Meetings.meetingsBuilder(inputTitle.getText())
        .startDate(logicApp.parseDateToString(inputStart.getValue()))
        .startTime(inputStartTime.getText())
        .endDate(logicApp.parseDateToString(inputEnd.getValue()))
        .endTime(inputEndTime.getText())
        .agenda(inputAgenda.getText())
        .build();
```

D) When instantiating a new Meeting, each mandatory value has to be passed into the constructor and each optional value can be set by calling the corresponding method and passing the value as a parameter

B. Observer Design Pattern

Motivation:

Implementation:

3. Unit Tests

- The project contains 15 unit tests from the Business Layer and Data Access Layer.
  I chose not to test the UI Layer as it does not really provide any valuable information and it would be difficult to test UI elements in unit tests anyways.
  The majority of the tests are made in the Business Layer as it contains the most important logic because it interacts with both the Data Access Layer and the UI Layer so every method needs to function perfectly in order for the application to work correctly.
  - ***CRUDLogicMeetingsTests***
    - Three tests in this method are testing Create, Update, Delete for the meeting object in the Business Layer.
    - All tests are aimed at the Validator method in CRUDLogicMeetings, which checks the input for mistakes, as to not flood the database with test data if I were to test the CRUD functionality in the Data Access Layer
    - Each test has two test cases – one with successful answer code and one that tests error detection for a specific mistake, to assure the Validator works correctly and sends through valid data, while catching any incorrect data input
  - ***CRUDLogicNotesTests***
    - Very similar to the above Meeting tests, there are also three tests for Create, Update, Delete, each with a correct input and incorrect input
    - There are slightly different criteria for input data completeness in the Note validator, therefore there are different error cases as in the Meeting tests
    - Each test case has a comment which describes what it is testing
  - ***AppLogicTests***
    - Here there are tests for the Read functionality from the Meetings and Note CRUD methods as well as a test for the PDF generation Validator
    - It is important to test that the Read methods are indeed able to connect to the database and fetch a list of entries as the list of meetings and notes is the most important part of the UI
  - ***ConfigurationLogicTests***
    - Here there are tests for fetching of data from the configuration file
    - It is important to test that these are read correctly, as it is easy to make a typo or a similar mistake while changing the configuration

entries, which would cause the entire application to be cut of from the database

- ***DBTests***
    - This section contains the only two tests which solely test the Data Access Layer, they test creating a connection to the application database
    - Thre is also a test for the checkUpdate() method, which lets the Business Layer know if an entry with corresponding properties was found in the database or if there are no such entries, this is important so the user gets correct feedback for his actions, otherwise he would not be able to tell that he tried to update a non existent Meeting or Note.