SE2-Projekt: Dokumentation

1. Meta-Data

Project: Sudoku

Group Members:
Lisa Kohls lk210
Lisa Salzer ls228
Johanna Rauscher jr132
Antonia Herdtner ah247

Link to Git-Repository:
https://gitlab.mi.hdm-stuttgart.de/lk210/sudoku/-/tree/master

2. Start

Abstract:

Our project is a sudoku game with three difficulty levels. Once the program is run, you can press *Play* to start a game. You then get to choose a level from 1 to 3 and will receive a Sudoku of a difficulty level accordingly. To insert the number into the grid, the player has to select the field in which a number is to be inserted and then choose the desired number from the number selection on the right side. If a wrong number is entered, the color of the field turns red and the *wrong input counter* is increased by 1. If the input counter reaches 3, the game is lost and the player is informed about that with a new window where he can choose to either reset the game or go back to the menu. The same options appear if the game is won. In this case, the *games solved* counter visible in the main menu is increased by 1. While playing, a game can also be reset, which will reset the *wrong input* counter and remove all numbers the player inserted from the game board. The *games solved* counter can also be reset by pressing the *reset score* button on the starting screen. Additionally to the counters, there is also a *time counter* that counts up the seconds needed to solve the game to make the game more interesting and competitive.

Main Class:

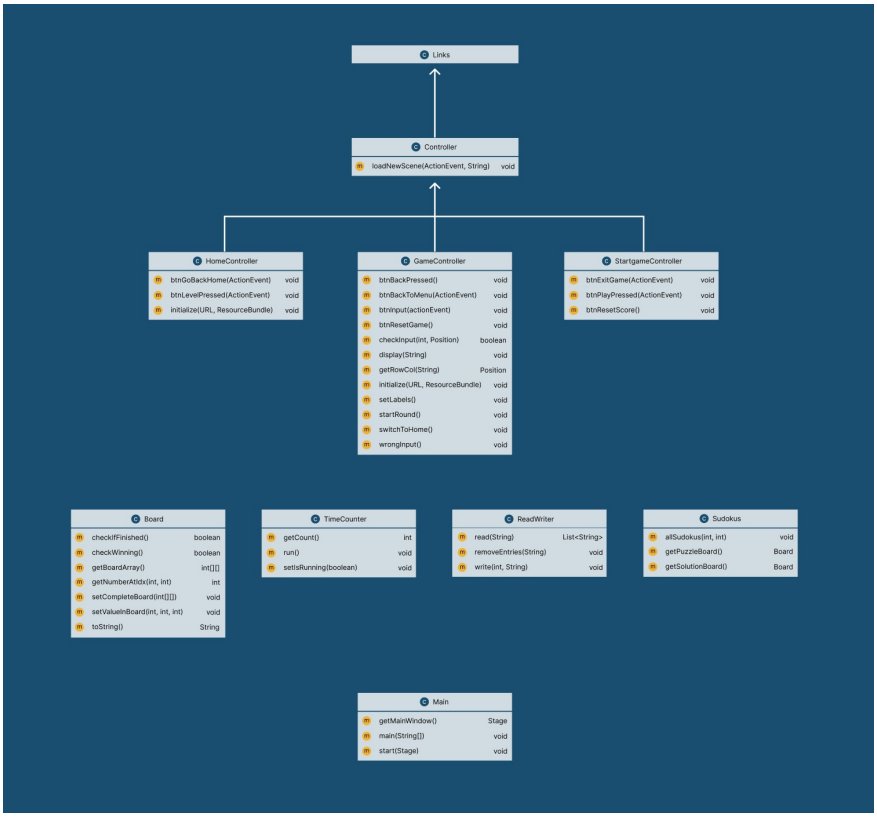
The main class is located at src/main/java/de/sudoku/game/Main.java

Besonderheiten:

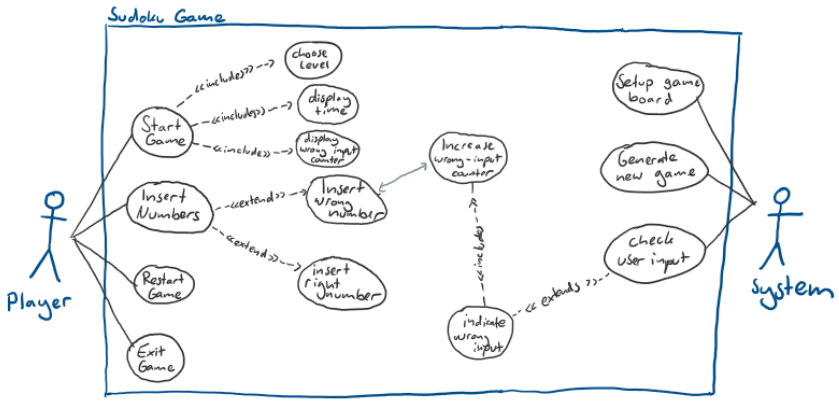
-

3./4. UML

Klassendiagramm:



Use-Case-Diagramm:



5. Assessment

Categories:

Architecture:

We used an abstract class for our file paths. So our Controller extends the abstract class *Links*. We used inheritance for our Controller classes. The classes *GameController*, *HomeController* and *StartgameController* inherit a method to load new scenes from the *Controller* class. In addition it includes our puzzle- and solution Boards as well as some Objects. We implemented the Interface *Initializable* from Java API.

Clean Code:

We have no static methods and we only implemented two static boards in the class *Sudokus* because they remain the same during the whole game process. Our getters don't return writable references to members. In the class *Sudokus* two getters return a solution board and a puzzle board. The method *getCount* in *TimeCounter* returns the current count.
A few more getters in *Board* class return the selected board. We have few public methods, most of them are private or protected. We stuck to the camelCase nomination. We used two packages to separate the FXML Controllers from classes to set the game. Our external files can be found in resources. Every FXML file has its own controller. We don't return important private objects which should be hidden. We have a memory-safe language. It controls the wrong user inputs by checking the number of wrong set values. If it's out of bounds for our restricted length, you can't finish the game and you will get a *log.error*: "Too many wrong inputs". In *gameController* we have a try-catch with an exception and logging error to prevent a runtime error when no label is selected and the user does an input. In general, our code is easy to read and understand as we used the java documentation and comprehensible naming of methods. One method has one task for example we have one method *setLabels* to set the labels and one method *startRound* to start the game.

Dokumentation:

We have many comments to describe the game process. We submitted our documentation.

Tests:

We have three test classes: *BoardTes*t to test some methods of the class *Board*, *ReaderWriterTest* to test the *ReaderWriter* class including one negative test and *TimeCounterTest* to check if a timer is not running while the game hasn't started yet.

GUI (JavaFX):

We believe that we fulfilled this point most successfully. We have four screens. Our first one is the start page with the option to reset your score, exit the game or start playing. If you press play a new scene will open with the current game score and the option to choose a level. The game score shows how many games were solved successfully in the past and can be reseted with the button on the start page. You can choose between three levels with different difficulties. If you press a level you can see the Sudoku board with different labels and buttons. All buttons are implemented with event handling. We included a css file for all scenes. If the game is finished, a new window will automatically open.
Which shows that we have a more complex GUI with a nested layout and multiple screens.

Logging/Exceptions:

We have set our logging level to debug. We have a logging framework in every class. We log exceptions (class *ReaderWriter*), threads (class *TimeCounter*) and log.info shows many game processes and user input.
We used try and catch when we are loading a new scene and when we use a file path in the *ReaderWriter* class for example. We intercept all possible exceptions for example when the user has no label selected and wants to add a number (*GameController*).

UML:

We have a clear and correct class diagram with explanatory effect and a use case diagram with the core classes.

Threads:

We have a thread in the *TimeCounter* class that counts the time needed for one game. In the class *GameController* thread.start() is called.

Streams and Lambda-functions:

We used some Lambdas for event handling of the buttons in the class GameController for example for the new window (display) “YOU LOST” or ”YOU WON”.