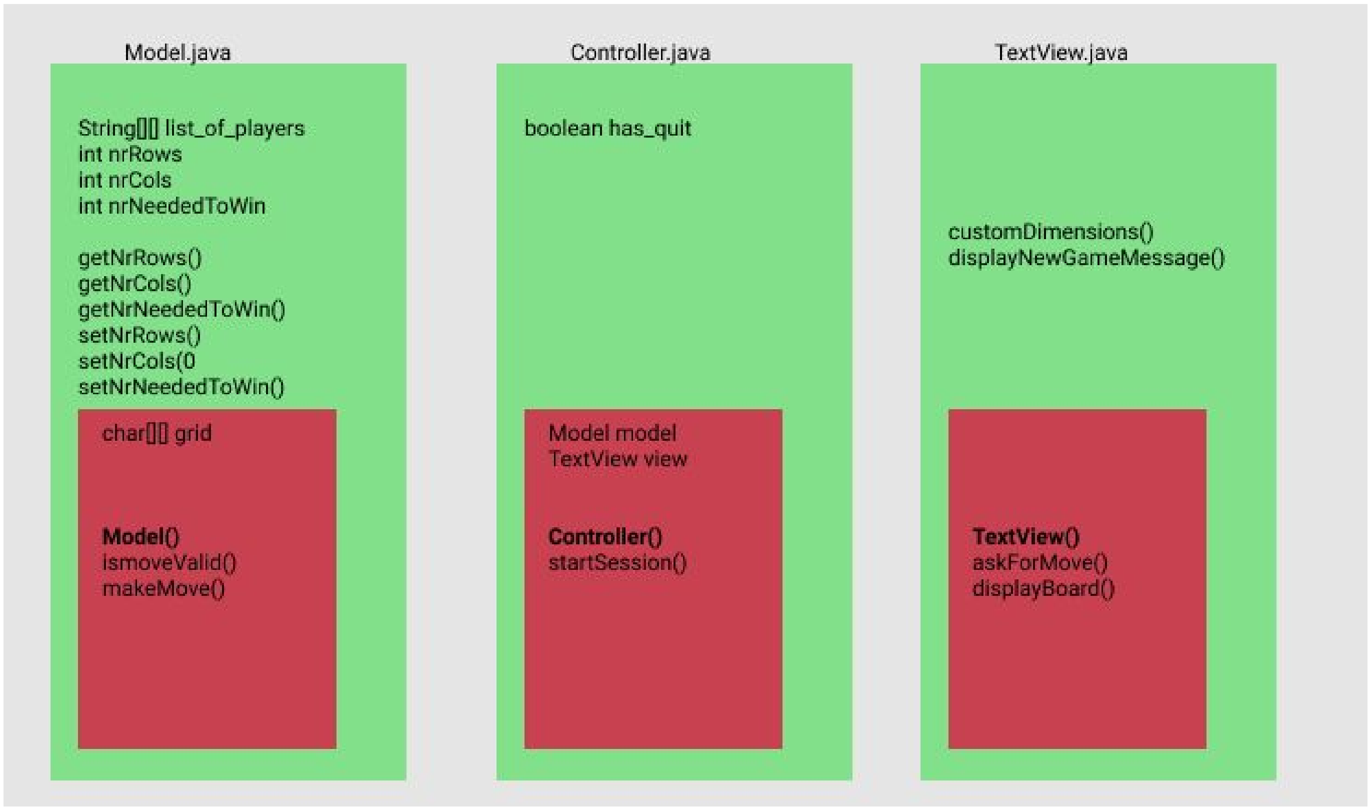
Assignment - 1 Reports

s2134605

Overview


Model.java
String[][] list_of_players
int nrRows
int nrCols
int nrNeededToWin
getNrRows()
getNrCols()
getNrNeededToWin()
setNrRows()
setNrCols(0
setNrNeededToWin()
char[][] grid
Model()
ismoveValid()
makeMove()
Controller.java
boolean has_quit
Model model
TextView view
Controller()
startSession()
TextView.java
customDimensions()
displayNewGameMessage()
TextView()
askForMove()
displayBoard()

Beginner features

The beginner step in this project sets out the basic code for the game. The implementation is very basic, rough and fragile. These problems of quality are addressed in the Intermediate expansion of the implementation.

Steps Done:

1. Creating a new project with the provided code
2. Familiarise yourself with the provided code
3. Model the state of the game
    - How was the board modeled: The board was modeled using a multidimensional array. The board could be modeled using a one-dimensional array as well. The strength of multi-dimensional modeling lies in the clarity and simplicity with which the board can be displayed with minimal manipulation. Modeling the board as a one-dimensional array makes the initial initialization of the array easy, but the subsequent necessary operations become needlessly complicated. It is for the sake of simplicity that the first was chosen.
4. Display the board
    - Board displayed using Stringbuilder
5. Allow pieces to be played
6. Implement the game loop
7. Game over
8. Complete any missing features
9. Write a short report

<u>Explanations for Key features:</u>

- How the program keeps track of which player goes next and how players are represented
    - The final String Array represents the list of players.

```
public static String[] list_of_players = {"player 1", "player 2"};
```

- To keep track of which player goes next, the game loop identifies which loop it is in. Since player 1 goes first, if the loop variable is equal to 0 or a multiple of 2, the makeMove method gets passed the user's column of choice, inputted using `InputUtil class` and the Character to display, 'R' for Red, 'Y' for Yellow.

```
for (int empty_cells = Model.getNrCols()*Model.getNrRows(), i = 0;
(empty_cells!=0 && !has_quit); --empty_cells, ++i) {
           if (i % 2 == 0) {
              System.out.println(Model.list_of_players[0]);
              view.displayBoard(this.model); //display the grid
      field of the model field of this controller object

              boolean valid;
              do {
                 int col = view.askForMove();
                 if (col == 0) {
                    has_quit = true;
                    break;
                 } //checking if user input is 0, which means they
      want to end current game
                 else {
                    valid = model.isMoveValid(col); //check if move
      valid, return boolean value
                    if (valid) this.model.makeMove(col, 'R');
                 }
              }
                 while (!valid) ; //while the input is not valid,
      keep doing this
           }

           else {
              System.out.println(Model.list_of_players[1]);
              view.displayBoard(this.model);

              boolean valid;
              do {
                 int col = view.askForMove();
                 if (col == 0) {has_quit = true; break;}
                 else{
                 valid = model.isMoveValid(col);
                 if (valid)  this.model.makeMove(col, 'Y'); }
              }
              while (!valid);

              }
```

```
            }

        }
    }
```

- Problems encountered and how they were solved
    - The only major problem encountered was trying to figure out how the program could figure out which character to display on the board when the player selects a column without requiring the user to input any more information.

        This was solved by abstracting the `makeMove` function using InputUtil.readIntFromUser() so that the user simply had to type in their selected column and did not have to call the `makeMove` method itself. This makes it appear that the `makeMove` method only has one argument, but it actually has two, one being the column the user selects, which the method receives through the Scanner sc, and the other being the character to display, which is selected based on the increment variable in the loop. See the above code snippet for details.

Intermediate features

Steps Done:

1. Allow the user to start a new game*
    a. In the game loop, the user is printed a query “New Game?”. To which the user can reply with a true for yes, or false for no. If the user answers true, there is a recursive call to the startSession function, if not System.exit(0) is used to quit the program.
2. Variable game settings*
    a. There is a method TextView.customDimensions(), which when called, asks the user if they desire custom dimensions, to which the user can respond using boolean values. If the answer is true, the method proceeds to ask them one by one, how many rows, columns, and marks needed to win. The input is taken in using the InputUtil readIntFromUser(). All these values are stored in an array, and the array is returned at the end of the method, since Java methods can only return one result. If the user did not want custom dimensions and answered false, the process of asking them about the dimensions is skipped and the default dimensions are returned in an array. In the startSession method, the result of the TextView.customDimensions() is stored in an array. Then the elements of the array are assigned to the nrRows, nrCols and nrNeeded_to_win. By having an else case if the user does not desire custom dimensions, we no longer need the DEFAULT_NR_ROWS and DEFAULT_NR_COLS variables.
3. Enhanced input validation*
    a. The method TextView.customDimensions() takes care of the input of the custom dimensions by using comparison operators to ensure that the arguments are valid. An example of this is ensuring that the number of pieces in a row that the user needs to win is less than the number of rows and number of columns. Further input validation is achieved through the InputUtil.java class.

Problems Encountered:

1. The expansion of the project to include the above features makes keeping track of exceptions and errors very difficult. Often variables are called and used in operations

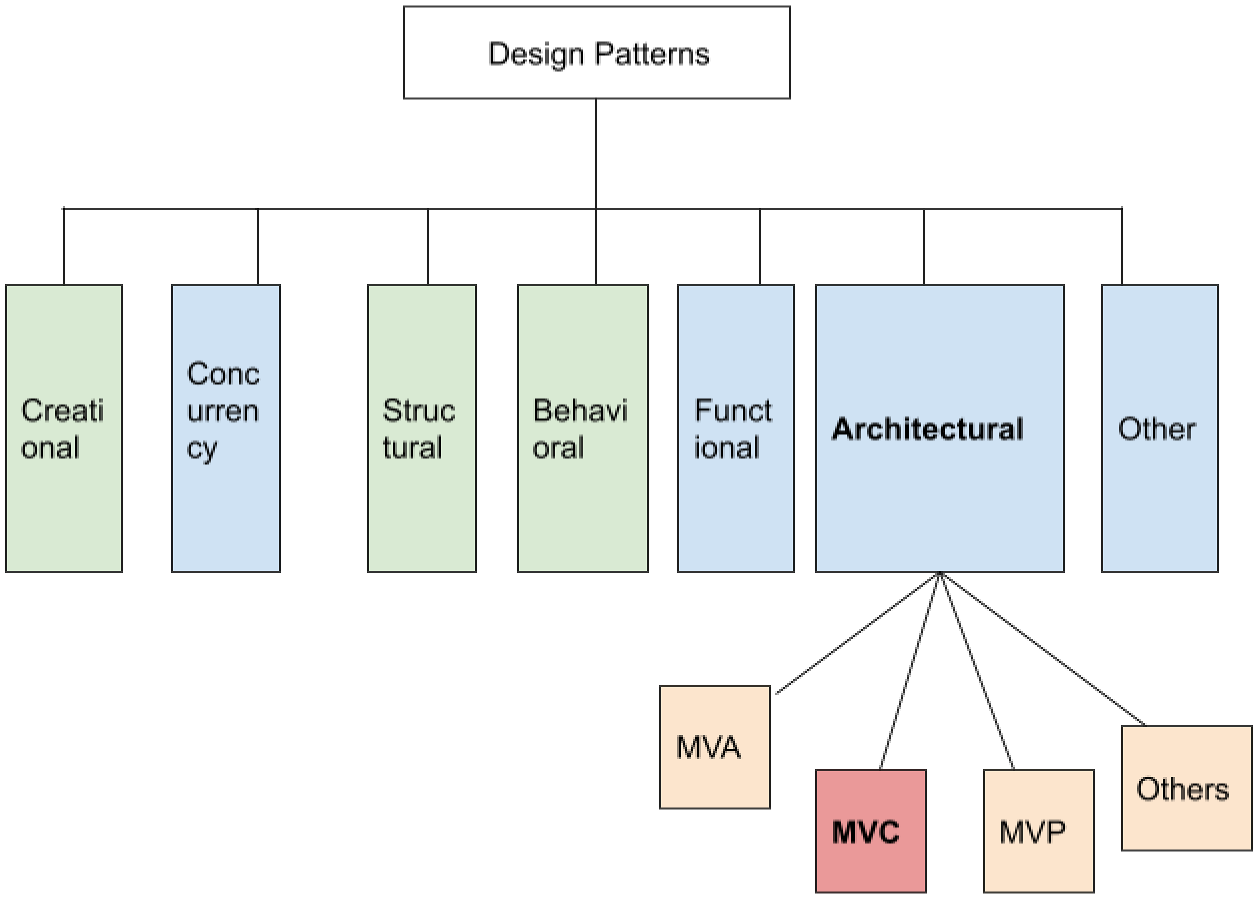
numerous times, making it difficult to understand what is exactly causing the problem. To counter this, I used IntelliJ’s debugger, setting breakpoints around the line that was identified in the error and exception message as causing the problem and stepping through to see exactly what was causing an exception to be raised. Most of the time the issue was isolated to a line or two, and the problem was not repeated throughout other files, but sometimes there was a problem in design and in this case I used the Refactoring process to go through and come up with a good solution.

Advanced Features

Steps Done:

1. Research a design pattern
   a. The MVC design pattern stands for the Model-View-Controller pattern. Widely used for GUI, it is now supported throughout many languages. This is a way of fragmenting the problem at hand into three interconnected parts (the Model, View and Controller).
      i. Model: The Model contains the logic and the data structures part of the project. In our project, the Model was represented by the Model.java class. The Model.java contained the model object, one of who's fields was the multidimensional array Grid, default constructor for the Model object, and methods that allow the Model to interact user by giving the user information with the TextView.java class and receiving input through the Controller.java class.
      ii. View: The View displays information to the user. In our project the View was represented by the TextView.java class, which contained the methods for asking the user for input, and displaying the various messages and the Connect4 grid.
      iii. Controller: The Controller is the primary way by which the Model receives information from the user. In our project, the Controller is represented by the Controller.java class, containing the StartSession method for starting the session.

   The MVC Design pattern is an architectural design pattern.



Design patterns can be divided into numerous sub classifications. Creational, Structural and Behavioral were the first design patterns. MVC is a sub classification of the Architectural Sub classification, which provides Structural organization for software programs.