Exercise_01

Ramona Walker, Dominik Johann Arnold, Mark Woolley, Otto Buck

January 2023

1 CRC Cards

Starting from the descriptions from the two appendices A and B we initially identified some candidate classes and wrote down their initial assumed responsibilities and collaborators.
From appendix A:

- Game. Takes care of the logistics and the game rules, sets up the game and keeps a record of who is playing and what the scores are. Decides if the game is over after a player played his turn.
- Player. Chooses a name, a color/symbol and an initial board size and configuration. Most importantly, she takes the decision which cell to kill and which one to create on her turn.
- GUI interface. Displays the game in a GUI which allows the players to take their strategic decisions.
- Terminal interface. Same as the GUI interface, but on a terminal.
- Computer/Program. The entity taking care of the game logistics, basically a synonym for game.
- Symbol/Color. Unlikely to be its own class, more of an attribute of the player class.
- Turn. Gives the current player the right to play her turn, taking the strategic decisions and updates the game state accordingly.
- Cell. Has to know if it's alive or dead. If the cell is alive, it must know which color it currently is of and also know if the state should change when a new generation takes place, based on the neighboring cells.
- Generation. Goes over all the cells and determine for each simultaneously if they change state in the new generation and if yes, of which color they are.

And from appendix B:

- Cell. Same as above.
- Color. Same as above.
- Turn. Same as above.
- Generation. Same as above.
- Board/Grid. Consists of all the cells in the game. In the original idea, there is no border. For our GUI and terminal interface however, we must have a border and it makes sense to have a class aggregating all the cells.

Following those possible candidates we created CRC Cards:

The *Game* class serves as the main class for playing the game. The class takes care of the logistics.

Class:	Game
Responsibilities:	**Collaborations**:
Playing the game	Player
Determine winner	Grid
	Cell
	GUI
	Terminal

Grid is a 2D arraylist of *Cells*. Each cell knows its state and its next state. The reason that a cell must know its current and its next state separately is that under the hood each cell is updated sequentially and doing that before the next cell knows its new state would lead to a loss of information. By calculating the next generation in two loops, we make sure, that each cell gets updated correctly.

Class:	Grid
Responsibilities:	**Collaborations**:
Store cells	Cell
Generate next generation	

The *Cell* class is the fundamental building block of the game. In our implementation, the cell does not store its location in the 2D grid, it only has to know its state.

Class:	Cell
Responsibilities:	**Collaborations**:
Provide State	PlayerColor
Update State	
Die instantly	
Come to life instantly	

The *Player* class stores a players name and color.

Class:	Player
Responsibilities:	**Collaborations**:
Store name	PlayerColor
Store color	

Each *PlayerColor* has a color name and a symbol assigned.

Class:	PlayerColor
Responsibilities:	**Collaborations**:
Store name	
Store symbol	

We also planned to make classes *Turn* and *Moves* because move (CreateCell) will be used when making a turn as well as when we initialize the starting position.

Class:	Turn
Responsibilities:	**Collaborations**:
Decide if a player has won	Grid
Make moves	Player, Player
	Moves

Class:	Moves
Responsibilities:	**Collaborations**:
Create cell	Cell
Kill cell	PlayerColor

We wanted to make an input parser to be able to separate the tasks of the game setup and turn taking from the validation of input.

Class:	InputParser
Responsibilities:	**Collaborations**:
Validate input for player names and player color	Player
Validate input for dimensions	Grid
Validate input for moves	

Initially, we did not yet have an idea how the GUI should look like. Accordingly, our initial CRC card was relatively vague.

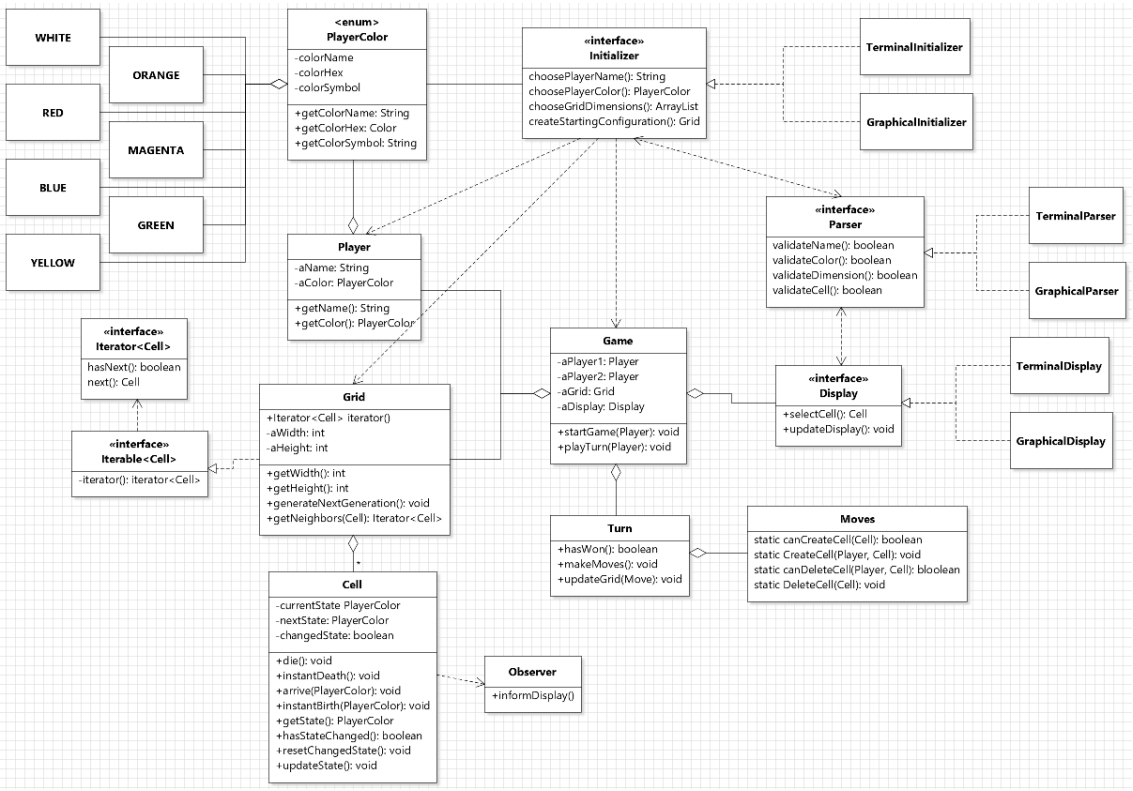
Class:	GUI
Responsibilities:	**Collaborations**:
Display Grid	Grid
Forward input to the parser	Game
	InputParser

An *Initializer* is used to set the game up. Two players will be assigned a color *(PlayerColor)* and a name as stated in appendix A.

We decided to let the starting configuration be chosen by picking 4 to 8 (configurable) cells which should come to live and then the board gets mirrored on the y-axis (when the player inserts the width of the grid, the whole grid will have twice the size of what is inserted initially). That way no matter the chosen grid size, both players will have a symmetric configuration.

Class:	Initializer
Responsibilities:	**Collaborations**:
Setup the game	Game, Player, InputParser, Grid, GUI

The following class diagram was created at the beginning of our project. When compared to the class diagram below one can see, that it has changed while programming.



2 Following the Responsibility Driven Design, describe the main classes you designed to be your project in terms of responsibilities and collaborations; also draw their class diagram.

From those initial identified candidates we finally decided not to use a separate turn class but to let that be implemented in the game class directly. The implementation of the GUI and Terminal ended up creating two different main classes which orchestrate the game. For the terminal version it is *TerminalMain* and for the GUI version it is *GameModel*.

- **Cell**
  A Cell is the smallest unit of the grid.
  Responsibilities: A cell has to know its current color. It has to be able to upgrade its color to the color of the next generation. For the GUI it also has to be able to register observers who are interested in a change of the state and notify those observers accordingly.
  Collaborations: *PlayerColor* for its state and next state. *GUIInitializer* and all the GUI elements that observe the cell.

- **Grid**
  The Grid is a collection of cells. It also knows where which cell is located.
  Responsibilities: A grid stores its dimensions and cells. It provides access to cells and can generate the next generation.
  Collaborations: *Cell* to construct the grid and update the grid. *PlayerColor* to upgrade cells.

- **GameModel**
  GameModel organizes the game by having the main components as attributes and playing turns. GameModel is only used if the game is played with the GUI. It serves as an observer as well as an observable depending on which GUI element is concerned. For the GUI elements that observe the GameModel, it must be able to notify those observers.
  Responsibilities: Store players, current player and the grid. Decides if a player has lost or won. Play turns.
  Collaborations: *Grid* and *Player* as attributes to play turns. *Moves* and *Cell* to execute moves. A *GUIInitializer* to initialize the game instance.

- **GUIInitializer**
  GUIInitializer is used to initialize the game with the GUI. It consists of a parser that makes sure no invalid input is accepted and gathers all the necessary information (names, colors, grid size, initial configuration) via GUI that are needed to set up the game instance. Like the GameModel, it serves as an observer as well as an observable depending on which GUI element is concerned. For the GUI elements that observe the GUIInitializer, it must be able to notify those observers.
  Responsibilities: Gather all information necessary to set up a game instance.
  Collaborations: *Players*, *Grid*, GUIInitializer observers to initialize the game. *InitializerParser* to check the input

- **TerminalInitializer**
  Like the GUIInitializer, the TerminalInitializer consists of a parser and is responsible for collecting all the necessary information for the TerminalMain to set up the game. Initializes the game via console (coordinates, names,...).
  Responsibilities: Scan input stream and give output for the console. Initialize the grid and players via console. Gather all information necessary to set up a game instance.
  Collaborations: *Grid*, *Player* and *PlayerColor* to initialize the grid. *InitializerParser* to check the input.

- **Moves**
  Moves contains static functions executing the game moves the player selects each round.
  Responsibilities: Kill cell and create cell.
  Collaborations: *Cell* to kill cells and create cells. *PlayerColor* to check if the cell is already dead/alive. *Player* to chose the correct color when creating.

- **InitializerParser**
  Validates all input made by the players.
  Responsibilities: Validate input.
  Collaborations: *GUIInitializer* and *TerminalInitializer* to check if name or color is already taken by player. *PlayerColor* to check if cell can be killed or revived and to assign valid colors to the players. *Grid* to check the coordinates.

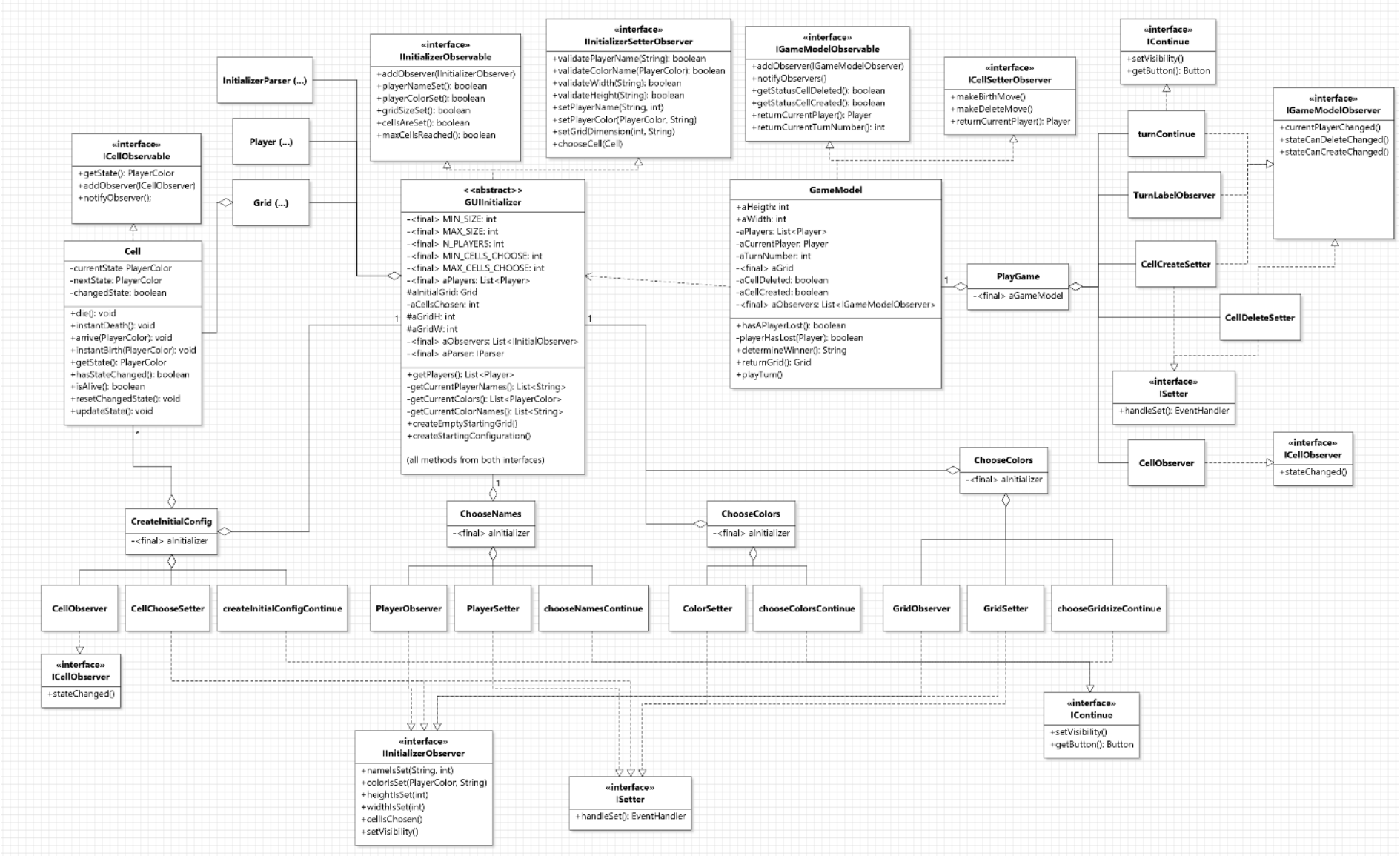
- **Player**
  The players store a name and a PlayerColor.
  Responsibilities: Return the name and the PlayerColor
  Collaborations: *PlayerColor* to return the PlayerColor.

- **PlayerColor**
  An enum type for colors. Each color has a name, a hex-code and a symbol.
  Responsibilities: return available colors (for the players to chose) and name, hex-code or symbol of a given color.
  Collaborations: none

- **TerminalMain**
  TerminalMain organizes the game by having the main components as attributes and playing turns. TerminalMain is used instead of GameModel if the game is played in the Terminal.
  Responsibilities: Store players, current player and the grid. Decides if a player has lost or won. Play the game.
  Collaborations: *Grid*, *Players*, *Cell*, *Parser*, *Moves*, *PlayerColor* to play the game. With *InitializerParser* to validate the input.

3 Draw the class diagram of the main classes

We split the class-diagram in two: one for playing with the GUI:



and one for playing with the terminal