Exercise_01

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

November 2022

1 CRC Cards

We created the following CRC cards before we started programming. We wanted *Game* to be the central class, which serves as connection between the classes and runs the game.

Class:	Game
Responsibilities:	**Collaborations**:
Playing the game:	Player
	Grid
	Turn
	Display

Grid is a 2D arraylist of *Cells*. Each cell knows its state and its next state. This is done, since if we update cells directly, we lose track of what the cell was before and cannot update them one by one. 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	Cell

The *Cell* should be the building block of the game.

Class:	Cell
Responsibilities:	**Collaborations**:
provide State:	PlayerColor
update:	PlayerColor
die instantly:	PlayerColor
come to life instantly:	PlayerColor

The Players get their class to store the names and colors.

Class:	Player
Responsibilities:	**Collaborations**:
store Name:	-
store color	PlayerColor

Each *PlayerColor* has a symbol assigned to be able to play the game in the console as well.

Class:	PlayerColor
Responsibilities:	**Collaborations**:
enum to store name and 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 won:	Grid, Player
make moves	Player, Moves

Class:	Moves
Responsibilities:	**Collaborations**:
Create cell:	Cell, PlayerColor
kill cell	Cell

We wantet to make an input parser to be able to use design by contract.

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

When we made the CRC cards, we did not really know how the GUI should look like.

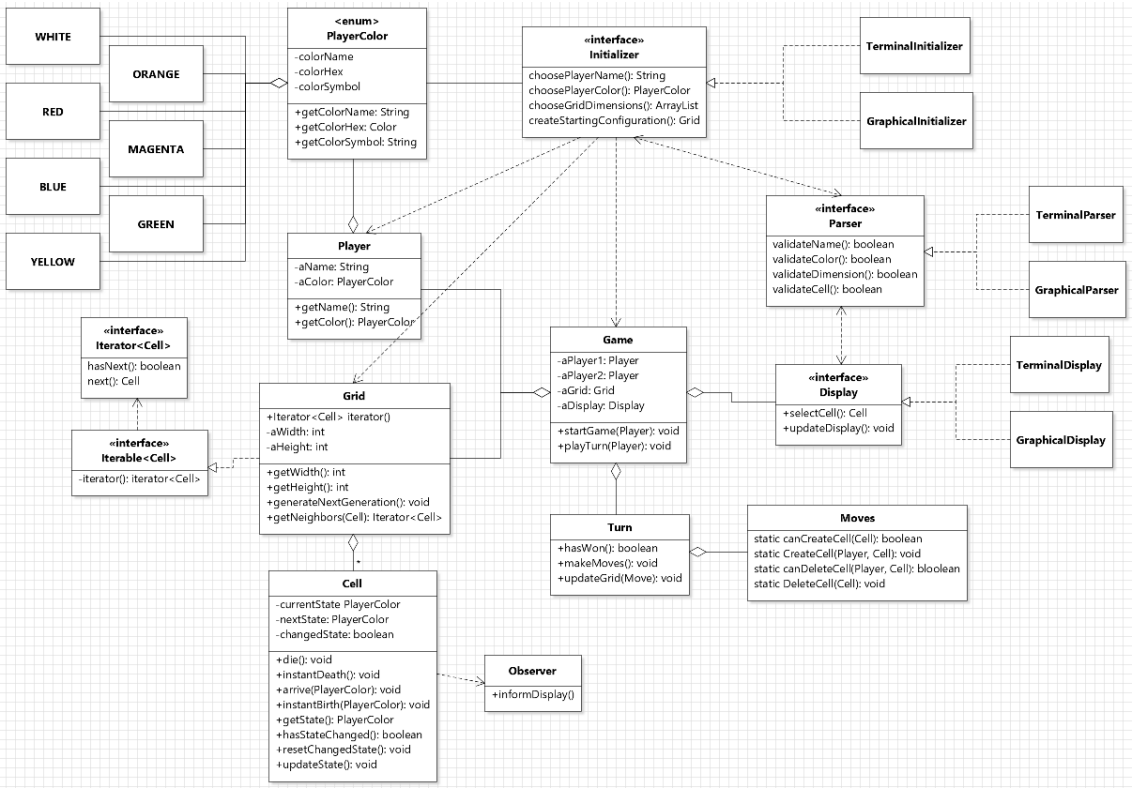
Class:	GUI
Responsibilities:	**Collaborations**:
Display Grid	Grid
forward input to turn	Turn

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

We decided to let the starting configuration be chosen by picking 4 to 8 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).

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.

- **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.
  Collaborations: *PlayerColor* for its state and next state.

- **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.
  Responsibilities: Store players, current player and the grid. Decides if a player has lost or won. Play turns.
  Collaborations: Grid and players as attributes. Cell to make moves. A Initializer to initialize the game.

- **GUIInitializer**
  GUIInitializer is used to initialize the game with the GUI. It makes sure that the size of the grid doesn't exceed the size of the monitor and creates observers.
  Responsibilities: Initialize game.
  Collaborations: Players, Grid, Observers to initialize the game. Cell to play turns. A Parser to check the input

- **TerminalInitializer**

Initializes the game via console (coordinates, names,...).
Responsibilities: Scan input stream and give output for the console. Initialize the grid and players via console.
Collaborations: Grid, Player and PlayerColor to initialize the grid. A parser to check the input.

- **Moves**
  Move has static functions executing the game moves the player does in each round.
  Responsibilities: Kill cell and create cell.
  Collaborations: *Cell* to kill cells and revivy cells. PlayerColor to check if the cell is already dead/alive. Player to give the right color to the cell when it is revived.

- **InitializerParser**
  Validates all input (check if coordinates in grid, player names...)
  Responsibilities: Validate input
  Collaborations: PlayerColor to check if cell can be killed or revived and to assign valid colors to the players. Grid to check the coordinates.

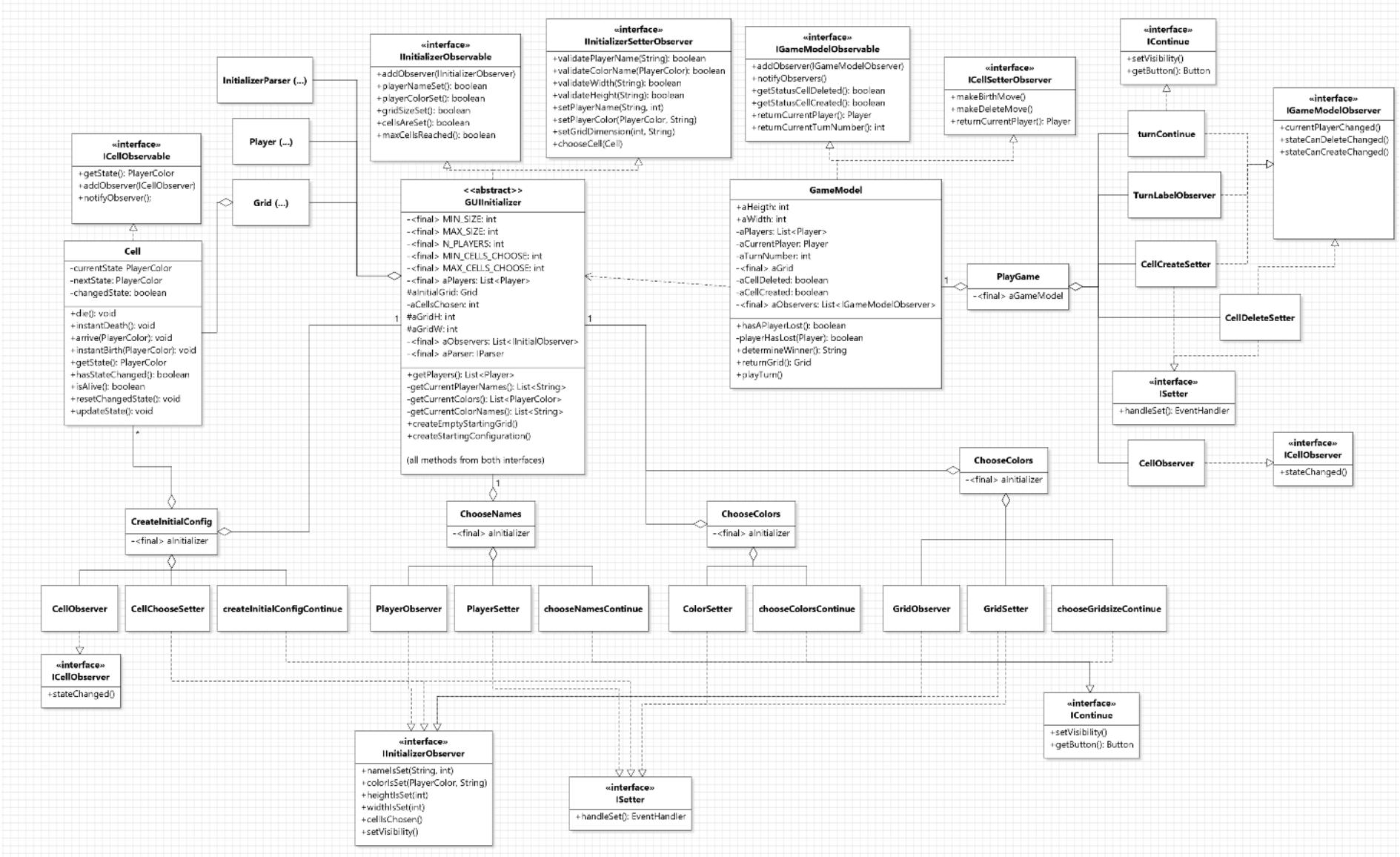
- **Player**
  The players store a name and a color.
  Responsibilities: The players have to be able to return their color and name.
  Collaborations: PlayerColor to return the color.

- **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: TerminalMain is the central class for playing in the console. It collaborates with Grid ,Players Cell, Parser, Moves PlayerColor and with an Initializer to set up and play the game.

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