Exercise_02

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

January 2023

1 Description of implementation decisions

1.1 Playing the game

The game can be played either in the terminal or a GUI. The GUI is implemented with JavaFX, it is therefore necessary to have JavaFX installed.
To play the game in the terminal, run the file *terminal.TerminalMain.java.*
To play the game in the GUI, make sure JavaFX is installed and the project settings have a library path set to the installation folder of JavaFX. Then, run the file *gui.GameOfLife.java.*

1.2 Initial configuration

The decision on the initial grid configuration on which the two players start the game must be decided during the initialization by the players. We think the rules and appendices are ambiguous regarding that aspect, only stating that *"The initial board configuration should be decided beforehand and be symmetric."*. The symmetry is achieved by letting the players decide on one half of the board and then mirror the board so that both players have an identical setup of alive cells. Regarding the *beforehand* aspect, we also discussed an option to let the computer decide on a random configuration which will be mirrored for both players. In the end, we decided to let the players chose the grid because that allows for much more interesting games.

In the terminal version, it's made explicit that the player who is first in alphabetical order gets to choose the initial grid configuration. In the GUI version, we decided to leave that ambiguous to let players decide together on one half of the grid, and the grid is mirrored on the Y-axis to create the final starting grid. In the GUI we decided to set limits to how many cells the initial configuration can entail. They are set to a minimum of 4 cells and a maximum of 8 cells. This can be changed in the *GUIInitializer.java* file in the fields *MIN_CELLS_CHOOSE* and *MAX_CELLS_CHOOSE.*

Both versions theoretically support an arbitrary size of the grid. This can be configured in for the terminal in *TerminalMain.java* with the fields *minHeight*, *maxHeight*, *minWidth* and *maxWidth* and in the GUI in the file *GUIInitializer.java* in the fields *MIN_SIZE* and *MAX_SIZE.*

1.3 Design decisions

- **Observer Pattern:** All the GUI elements make ample use of the observer pattern learned in class. During implementation we realized that many of the elements need to have a double role as observer and observed simultaneously. For example the *GUIInitializer* class is obviously the observed model that holds data for many of the display elements of the GUI (When a player name is set, the *GUIInitializer* must let the text field now that so that the text field can display the valid set name. The text field is the observer, the *GUIInitializer* is the observed). However, at the same time, the *GUIInitializer* also is an observer of the setter buttons in the GUI. As soon as those buttons are clicked (For example the Set button to set an entered player name), the button needs to let the *GUIInitializer* know that a player name has been set. That way, the *GUIInitializer* can react to the action of the set button by changing the data (player name set) internally, which then in turn triggers a callback to the text field observer to display the name.
- **Separation of concerns:** Instead of creating few big interfaces for all observers and observables, we decided to separate different observers into different interfaces. Otherwise, many of the implementing classes would only use a small fraction of the implemented methods of the interface.

- **Inheritance:** Some of the reused GUI elements like the continue buttons or the setter buttons have a superior abstract base class which simplifies the reuse of many of the elements for multiple different GUI stages. For example the setter elements have very similar behavior, except for some labeling. The abstract base class prevents code duplication by setting defaults for all subclasses.
- **Iterator Pattern:** We implemented the iterator pattern in the *Grid* class to iterate over the cells in the grid in different ways, one from top left to bottom right, one reversed.
- **Design by contract and input validation:** Where ever possible, we tried to implement design by contract or input validation by the parser class to check inputs from the users.
- **Encapsulation:** Where ever possible, we tried to keep exposing internal data structures and implementations to a minimum, letting classes expose only what is necessary to other classes. All fields are set to private where possible. That way we try to achieve a good encapsulation.
  We also tried to encapsulate abstractions, for example the *Player* and *PlayerColor* classes make use of enums to encapsulate the concept of a player with a name and a color as far as possible.