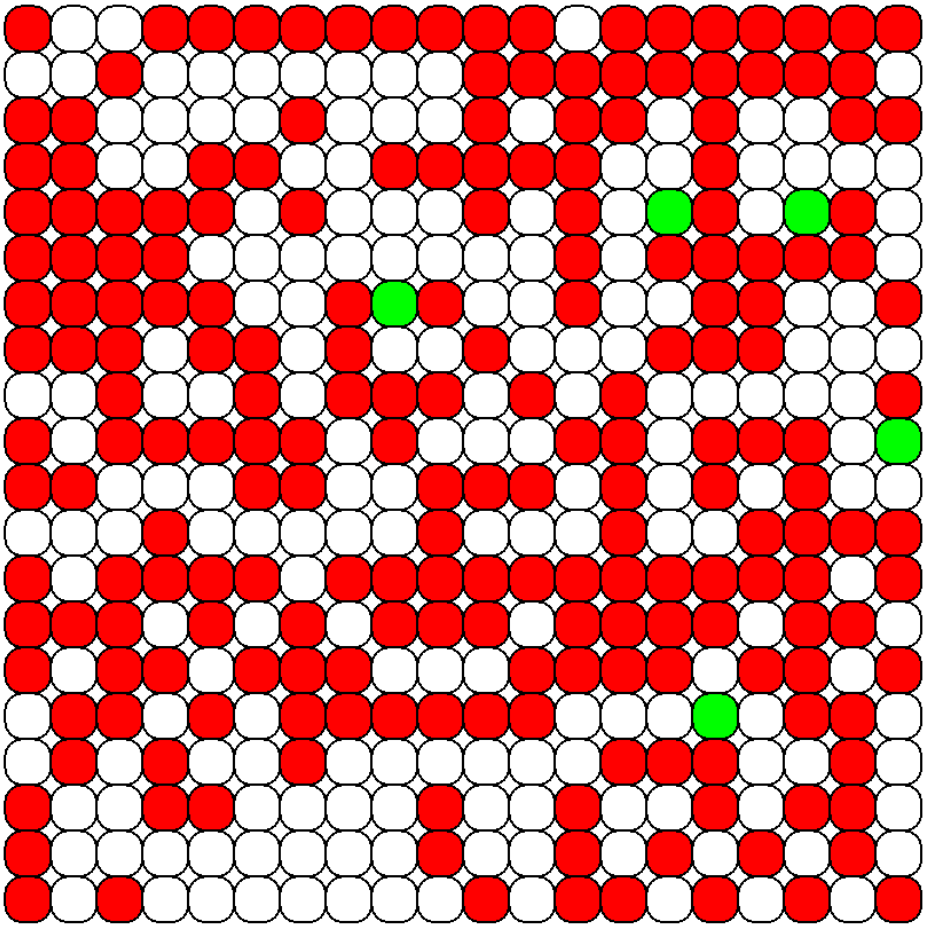
LP24 Spring 2020

Game of Life Space Version

Thomas Duvinage



utbm
université de technologie
Belfort-Montbéliard

Table of contents

Figures table

Game of Life

Origin

The Game of Life has been created by John Horton Conway in 1970, he was a mathematics professor at University of Cambridge, in the United Kingdom. J.H. Conway was interested in the problem proposed by John von Neumann. This problem consisted in finding a machine which can reproduce itself. He succeeds by creating a complex rules model in a cartesian coordinate system. Conway tried to simplify von Neumann ideas and succeeds. Coupling his success with previous successes, he created the game of life.

Figure 1 : John Horton Conway

The Game

Explanation

The universe of the Game of Life can be finite or infinite. It's represented by a two-dimensional orthogonal grid, each cell of the grid has a two possible state, alive or dead. This state is generally represented by setting a different cell color between both states.

Figure 2 : left grid is composed of dead cells and right grid is composed of one live cell in the middle

Every cell interacts with its eight neighbors, which are all that are around the designated one. This representation is called the Moore Neighborhood (Wolfram, Moore Neighborhood, 2020). This method is used to define a set of cells surrounding a given cell at a given position (x_0, y_0). With the Moore method, we can set a range value named r to get the set of cells which are inside this range.
Then the set of cells is defined by

$$N^M_{(x_0,y_0)} = \{(x, y) : |x - x_0| \leq r, |y - y_0| \leq r\}$$

The following Figure is an example of the Moore method.

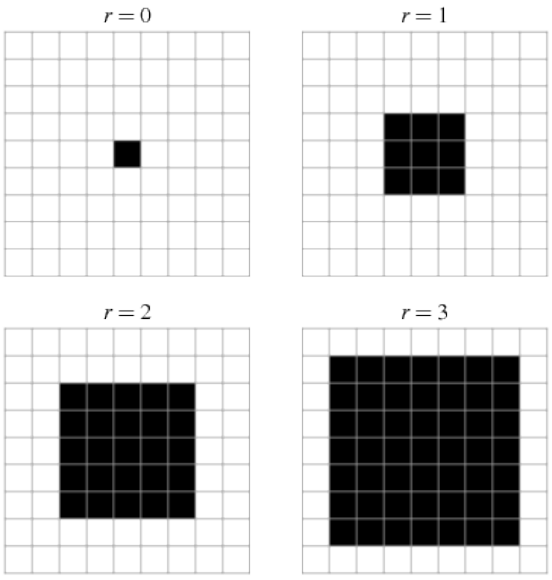


Figure 3 : Application Moore Neighborhood method @Wolfram

Then for each cell in the grid transitions occur after each iteration depending on the rules applied on each cell.

Rules

As explain in (Contributors, 2020), at each iteration of the game, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if by underpopulation.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

These rules, which compare the behavior of the automaton[1] to real life, can be condensed into the following:

- Any live cell with two or three live neighbors survives.
- Any dead cell with three live neighbors becomes a live cell.
- All other live cells die in the next generation. Similarly, all other dead cells stay dead.

To give you an overview of the result, here is an example of a simulation of the Game of Life :

1 A cellular automaton is a collection of "colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. (Wolfram, Cellular Automaton , 2020)

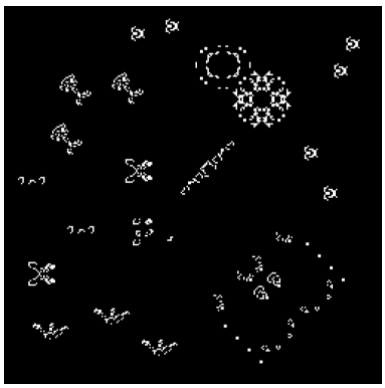


Figure 4 : Example of a game of life simulation

Patterns

The initial pattern constitutes the base of the game. To generate the initial pattern there are two possibilities: automatic generation or user completion. In the automatic mode, the initial pattern is generated by randomly applying a state to each cell of the grid. Then, after each iterations a new pattern is generate due to the application of the rules mentioned above.

Patterns are some things really amazing in the Game of Life. Due to the application of the rules mentioned above, it some situation the game can have some repeatable patterns. Which is something really cool. Below is a small set of patterns of the game of life. Many other patterns exist see here if you want to know more about them.

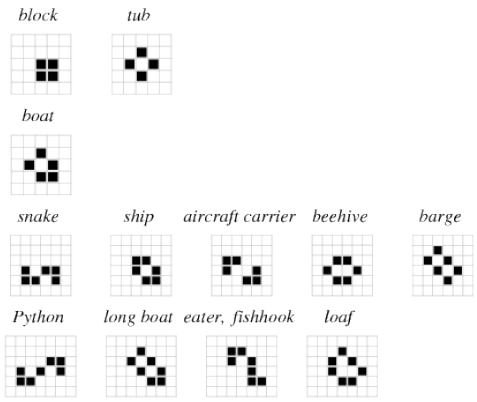


Figure 5 : Patterns example

Game of Life Space Version

Description

The Game of Life Space Version is the game I've created during this semester. The instructions for the project were:

"The goal of this project is to adapt the original game of life to a one-player version. This can take the form of a set of mini-games that you will have to define by yourself (for example, reaching a special pattern by choosing the right initial form, or permitting to the player to remove/add cells during the simulation, make several pattern fight each other, reach a certain score before the time is up, introduce bonus cells or a special mode with different rules, etc.). You can define several levels of difficulties that will limit the possible interactions or their amplitudes for the player.

You should pay attention to the undecidability of this game, and make sure that the player can in fact complete the mini-game or the level. You may choose to limit the grid to a certain size, or to make it evolve (under some constraints) during the game, depending on the moves and evolutions of the cells."

The Game of Life Space Version is a one-player game.

The grid is composed of three kind of cells:

- Blue cell: represents the player
- Green cells: represent the planets
- Red cells: represent the meteoroids

When the player presses the button play, the meteoroids are moving while respecting the rules of the Game of Life explained above.

The planets are static in the grid, the goal of the player it to conquer each planet by passing over them. When the player conquers one of them it disappears. When all of the planets are conquered the level is done.

Rules

The rules are simple, the player is represented as a blue cell in the grid, he can move in the grid by pressing the cross keys on his keyboard. The red cells are representing meteoroids, the player can't go over a meteoroid, he has to avoid the meteoroid to reach the planets. Planets are the green cells in the grid. The number of planets to reach depends on the level of the player, the number of planets equals the level of the player plus one. Then when the player reaches all the planet, he finishes the level and can then play a new level-up game or stop playing.

During the game the player can interact with the grid to add or delete cells. The goal of this add-on is to avoid blocking situation during the game. Furthermore, it can help the player to create patterns in the grid to complete the level.

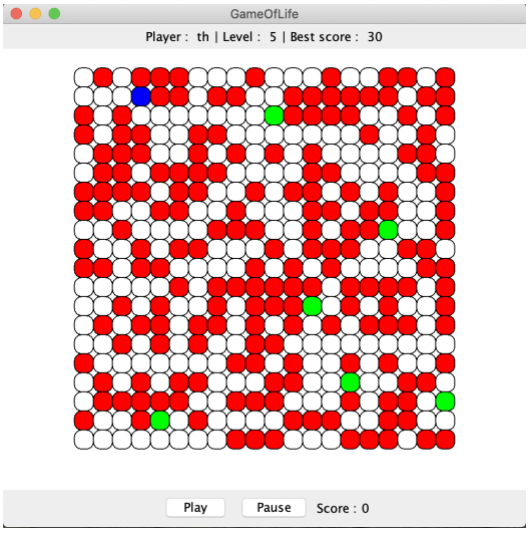


Figure 6 : Example of a game with the player th

How to play?

To play to the Game of Life Version, you have to run the executable GameOfLife_SpaceVersion.jar. After launching the executable, the starting page appears as in the starting page Figure below.

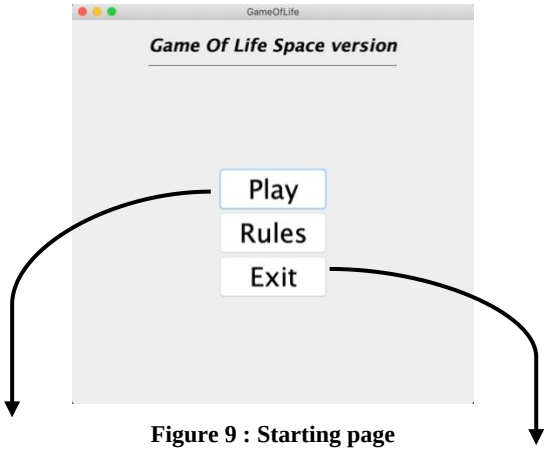


Figure 9 : Starting page

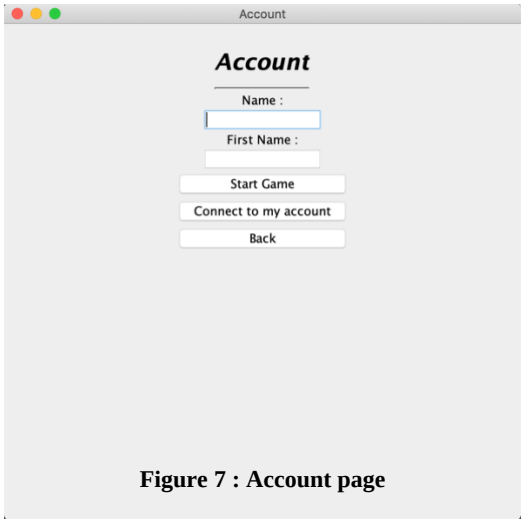


Figure 7 : Account page

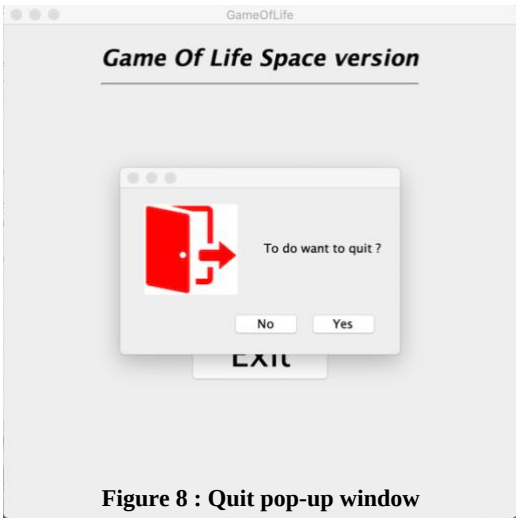
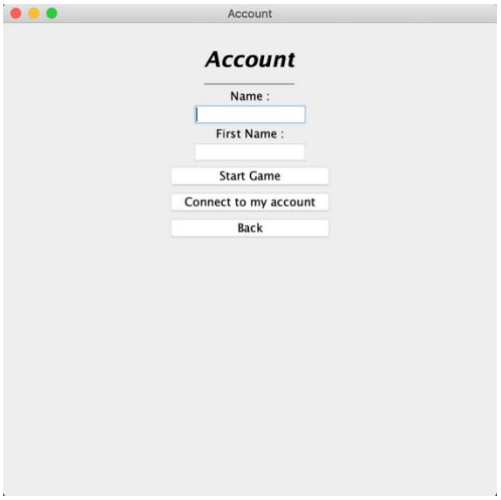


Figure 8 : Quit pop-up window

Play button

If the player clicked on the Play button, the Account page will appear.



Thus, two distinct situations emerge:

1. The player already owns an account
2. The player doesn't own an account

In the first situation, the player has to write down his Name and First name in the specific input box. Afterwards, he has to press the Connect to my account button.

In the second situation, the player has to create an account. To do so the player as to fill the input box with his personal informations. When the input boxes are filled the player can press the Start Game button, by pressing the button, the player account is created.

The algorithm and the storage of all the players will be explained in the Player section.

When the player logs in, the game's window appears as in the Figure below.

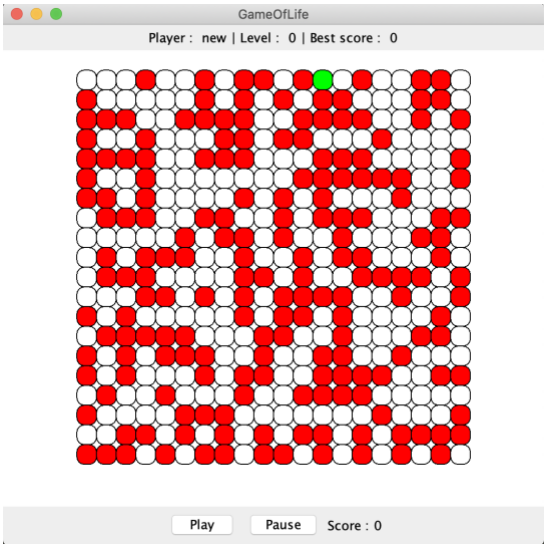


Figure 10 : game window

Rules

The Rules button is used to let the player know the rules of the game.

Exit button

The exit button can be used by the user to quit the game. A pop-up window appears to ask the player to validate his choice as in the quit pop up window Figure.

MVC

Model-view-controller also known as MVC is a software design pattern commonly used for developing user interfaces. The goal of this method is to divide the related program into three interconnected elements. This is used to separate the information from the way information are presented to the user. This method is like implementing different layers of a page.

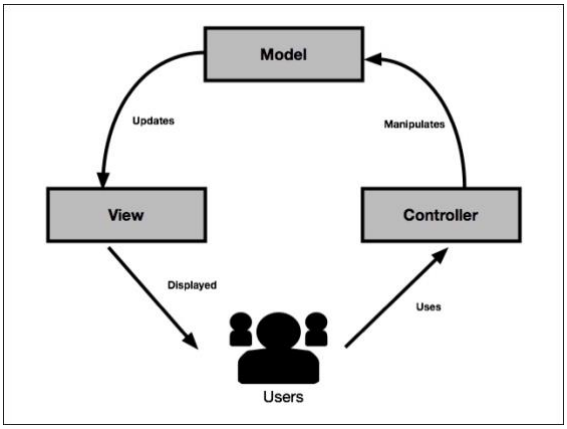


Figure 11 : Model-view-controller (MVC) pattern
@packtpub

Model

The central component of the pattern. It's the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

View

Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

Controller

Accepts input and converts it to commands for the model or view.

In addition to dividing the application into these components, the model–view–controller design defines the interactions between them.

- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view means presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

Dealing with this new principle has been something new for me this semester and I'm really happy to know how to deal with it know knowing that it will be very useful in my future project.

The whole structure of the project is based on this pattern. When I started to code the project, we hadn't studied it, so my code was messy, and problems started to appear. Then I switch to the MVC pattern and it has been a way easier method to implement the project.

OOP

Structure

My project is made of four packages.

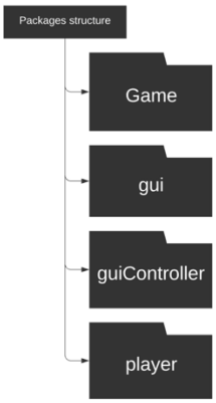


Figure 12 : Packages structure

- Game: regroups all the classes which are representing the "basic" classes of the game such as Cell, Grid, Point2D.
- gui: regroups all the classes which are representing all the views of the game.
- guiController: regroups all the classes representing the actions all linked to a button or events listeners. Those classes interact with the classes containing the gui package mentioned above.
- player: regroups all the classes representing the player. In contrary to the organization of other packages, this package contains all the classes concerning the view, model and controller. To make it understandable, I have given a clear name to each class.

Now that you know the package structure, I'm going to explain to you the classes and their implementation in each package.

Packages

Game

The game package is composed of three classes: Cell, Grid, Point2D.

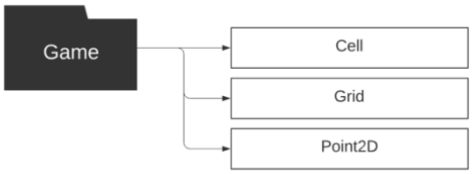


Figure 13 : Game package classes

Point2D

This class is used to represent the position of a cell in the grid. I decided to implement my own Point2D class than using java.awt.geom.Point2D. This was a way for me to control the whole project.

Then the Poin2D class has two attributes:

Figure 14 : Point2D public attributes

Cell

The Cell class represents each cell of the grid. Each cell has private attributes:

Figure 15 : Cell private attributes

As you can see on the Figure above, the cell has 4 different states.

- State: actual state of the cell, this state is updated after each iteration and is replaced by the newState attribute.
- newState: state of the cell after applying the rules on the grid.
- playerState: each time the player moves on the grid, the playerState of the cell representing him is updated to true, otherwise it is false. When the player leaves the cell, it becomes a usual cell and can then become a meteoroid.
- planetState: when generating the initial game pattern, planets are created, then each cell representing one planet has a planetState to true, otherwise it is set to false. When the player is passing over the planet the planetState of the cell is set to false, then the cell can become a meteoroid.

The Cell class inherits from the Point2D class which is explained below.

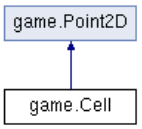


Figure 16 : Cell inheritance

Another possibility would be to create a planet and a PlayerCell class. These classes would inherit from the cell class, then override the draw method and override the getState method. This allows doing polymorphism. I did not do this because I realized it too late in the implementation of the project and given the time I had left; I made the choice to keep the current configuration but I am completely aware that a structure using polymorphism was possible.

Grid

This Grid class is used to handle the game's grid.

The grid has many private and public attributes.

- ***Public***

Public Attributes

int	**width** width of the grid More...
int	**height** height of the grid More...
int	**cellWidth** Cells width. More...
int	**cellNumberWidth** Cells width number which is computed depending on the cellWidth. More...
int	**cellNumberHeight** Cells height number which is computed depending on the cellWidth. More...
int	**gridPosX** X Position of the grid in the Panel. More...
int	**gridPosY** Y Position of the grid in the Panel. More...

Figure 17 : Grid public attributes

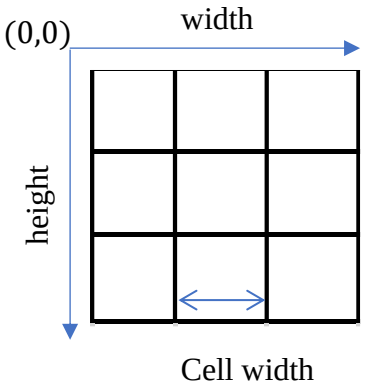


Figure 18 : grid dimensions

The width, height and the cell width of the grid are set by passing them into the constructor. Then with those parameters, the cellNumberWidth and cellNumberHeight can be computed by dividing the width and the height by the cellWidth.

- **Private**

Private Attributes

Cell[][]	**grid** 2d array array of Cells representing the grid More...

Figure 19 : Grid private attribute

Gui

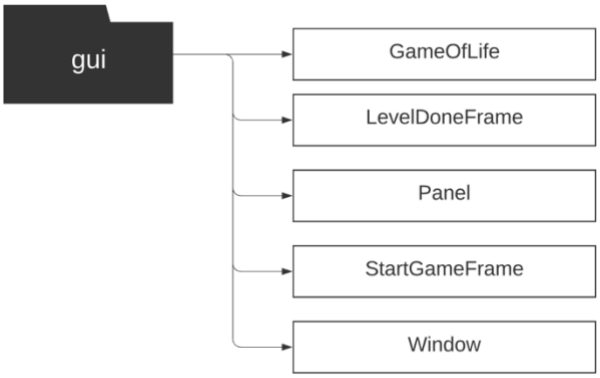


Figure 20 : gui package classes

GameOLifePackage

This class is the main class the game, if you want to play to the game, you have to run this class. Before playing, make sure to read the entirety of this report because you'll have to install the JSON library to make the game works great.

LevelDoneFrame

This class inherits from JFrame. This frame is displayed when the player completes a level. It shows nformation on the player's level and ask the player if he wants to play again.

Figure 21 : Level done frame

Panel

This class inherits from JPanel. This panel is used to handle the grid. This panel is instantiated in the Window class.

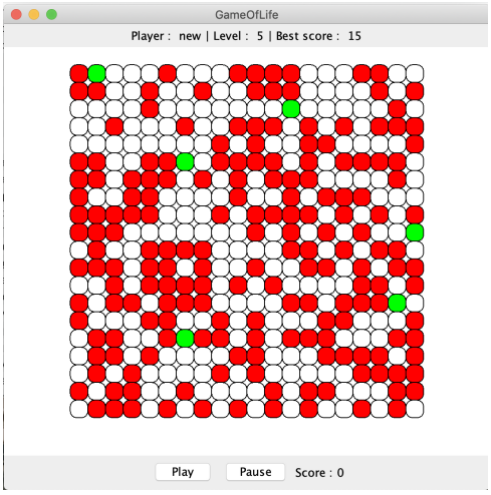
StartGameFrame

This class inherits from JFrame, it is used to represent the start window which contains the menu button.

Figure 22 : Starting page

Window

This class is one of the most important class of the project because it's the one that handle the class Panel (grid) and the play, pause button. To do so the Window class inherits from JFrame.



Information about the user are displayed at the top of the Frame.

You will remark that there is not a back to menu button. Indeed, I've chosen to do so because if the player wanted to come back to the main menu, he can simply quit the game, so I figured that if he wants to leave the game, he only has to press the close button.

GuiController

The guiController package contains all the classes used to manage event since the starting page is launched. These classes manage the button, mouse and timer events. To do so all of them at the expectation of the TimerMainWindowController class, implement the ActionListener interface when the class is used to handle button events or MouseListener interface if the class is used to handle mouse events. An interface is an abstraction of class with no implementation details, which means that the methods are declared but not implemented. Then methods have to be overridden to implement the content.
Compare to subsections above, no details will be given for each class as they are quite similar.

The only difference is for the TimerMainWindowController class which is used to handle the event since the play button has been pressed. The event consists in calling the update method on the grid. Which consists in applying rules on the current grid and then repaints the Panel which contains the grid. This event is scheduled and append at a given rate specified when calling the schedule method.

```
timer.schedule(new MyTask(), 500, 1000);
```

This line is meant to create a new MyTask() during 500 milliseconds every second. MyTask class consist in the method mentioned in the paragraph above.

```
public class MyTask extends TimerTask {
    @Override
    public void run() {
        gridPanel.grid.update();
        gridPanel.repaint();
    }
}
```

Player

Has I mentioned before a package is dedicated to the player. This package contains the view, model and controller classes. To avoid misunderstanding, explicit classes names have been chosen. In this section I'm going to detail all the proprieties and actions of the player. At the end, an explanation of the storage of the players will be given.

Proprieties

The player has multiple attributes.

Figure 23 : Player attributes

Actions

As I said above, the player is represented in grid the grid by a blue cell. The player can move in the grid using his arrow keys on the keyboard.

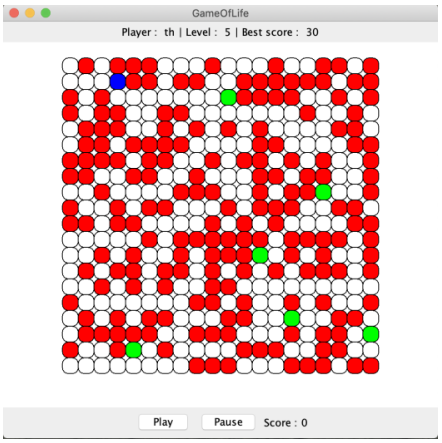


Figure 24 : Player in the grid

Each time the player press an arrow key a position buffer is created the goal of doing this is to call the setPlayerCell() method.

```
/**
 * @brief
 * @param playerPos
 * @param previousPlayerPos
 * @return
 */
public boolean setPlayerCell(Point2D playerPos, Point2D previousPlayerPos) {
    if(playerPos.posX >= 0 && playerPos.posX < this.cellNumberWidth && playerPos.posY >= 0 && playerPos.posY < this.cellNumberHeight) {
        if(!this.grid[playerPos.posY][playerPos.posX].getState()) {
            this.grid[playerPos.posY][playerPos.posX].setPlayerCell();
            this.grid[previousPlayerPos.posY][previousPlayerPos.posX].setDefaultCell();
            return true;
        }
    }

    return false;
}
```

This method checks if the player goal cell is a meteoroid or not and if it is in the game boundaries. If it doesn't respect those conditions, the player cannot move, and the method returns false. Otherwise, if the cell is in the boundaries and isn't a meteoroid then the cell is set to a player state. The previous cell occupied by the player is set to a default state set which means that the cell can become a meteoroid.

Save

To save every players information, I've decided to use a .json file.

Why json?

JavaScript Object Notation mainly known as json is an open standard file format, it's mostly used in web development. JSON is really useful to exchange data or store information because the information stored into it are easy to access. We can compare it for instance to a dictionary in python of hashMap in java. Furthermore, all kind of data can be stored into a json file. Let's look at an example:

Figure 25 : player.json file

The two lines above is an example of the player.json file used in the game. All the content in the braces is all the information about one player.

The information stored are:

- First name
- Last name
- Level
- Best score

To access the data, you have to iterate over the array, then for each cell of the array, you can access the player information by using the JSON-simple library get method. To do so, you have to create a JSON object obj for each cell of the array and then use the get method by doing obj.get("firstName"), the string representing the first name of the player will be returned.

When a player sign-in the game, a new JSON object is created and added to the player.json, like that, the player can keep his profile.

When the player finishes a level, the JSON object representing him in the player.json is updated. He can leave the game and come back to play again at the same level he was before.

JSON-simple-1.1.1 library

To deal with JSON in the project I've had to add the library to the java project in Eclipse because this library isn't natively included in the Java VM.

To do so, you have to right click on the *Referenced Libraries* folder.

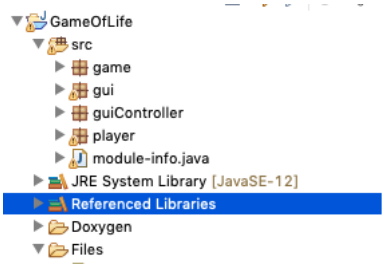


Figure 27 : Referenced libraries folder

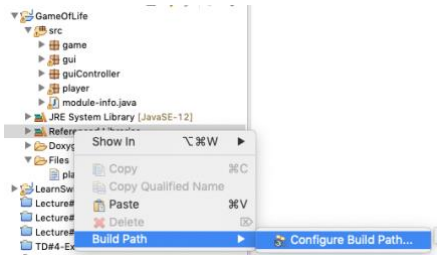


Figure 26 : second step add json library

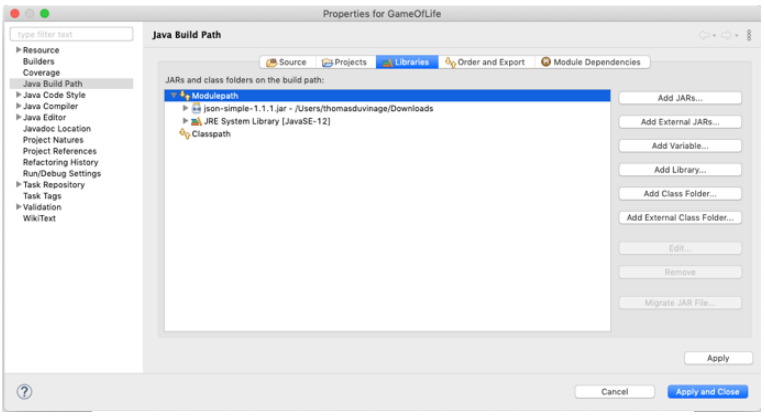


Figure 28 : Third step add json library

When the configured build path window is opened, click on *Add external JARs…* and then add the json-simple-1.1.1.jar given is the folder. When it's done the library might be in the *Referenced Libraries* folder.

Issues

The project isn't working in its integrity. In fact, when the player login or sign up, a window with the grid into it appears. Before the player press on the start button, he can move his cell. But since the player pressed the play button at the bottom of the window, he cannot move.

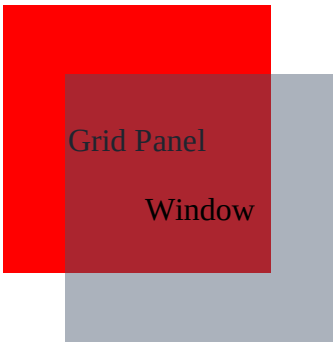


Figure 29 : Window organization

At the beginning the problem was that the focus wasn't made on the window, so I added the following line: Window.setFocusable(true) and one problem was solved. The problem of focus is specified in (Oracle, Class KeyEvent, s.d.) which says that *"As specified in Focus Specification key events are dispatched to the focus owner by default."*

After adding this line, the player was able to move in the grid before pressing the play button, but not after.

Here is the biggest trouble and sadly what's blocking me to finish the project.
To make the grid evolved over the timer, I decided to use a Timer. As explained above, the Timer is used to call a method at a given rate. Since the player presses the play button, the Timer is instantiated. At this precise moment, the player cannot move in the grid anymore.

Then I asked myself about the method used to make the player move in the grid. The first method was KeyListener. The problem was still the same, the focus was made on the Window but not on the Grid Panel then the KeyEvents are sent to the Window, but the events have to append on the Grid Panel.

After hours of debugging, I read that I could use key binding. In (Oracle, How to Use Key Bindings, s.d.), it says :*"Key listeners have their place as a low-level interface to keyboard input, but for responding to individual keys key bindings are more appropriate and tend to result in more easily maintained code. Key listeners are also difficult if the key binding is to be active when the component doesn't have focus. Some of the advantages of key bindings are they're somewhat self-documenting, take the containment hierarchy into account, encourage reusable chunks of code (Action objects), and allow actions to be easily removed, customized,*

or shared. Also, they make it easy to change the key to which an action is bound. Another advantage of Actions is that they have an enabled state which provides an easy way to disable the action without having to track which component it is attached to."

Key binding involves two objects, InputMap and ActionMap. InputMap maps user input to an action name, ActionMap maps an action name to an Action. When the user presses a key, the input map is searched for the key and finds an action name, then the action map is searched for the action name and executes the action.

Here is an example of how to use key binding:

```
myComponent.getInputMap().put("userInput", "myAction");
myComponent.getActionMap().put("myAction", action);
```

What was for me the most import feature of key binding is that InputMap is reacting to different focus states. As explained in (Oracle, How to Use Key Bindings, s.d.), each JComponent has one action map and three input maps. The input maps correspond to the following focus situations:

- JComponent.WHEN_FOCUSED
  - The component has the keyboard focus. The WHEN_FOCUSED input map is typically used when the component has no children.
- JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
  - The component contains (or is) the component that has the focus.
- JComponent.WHEN_IN_FOCUSED_WINDOW
  - The component's window either has the focus or contains the component that has the focus.

In my case, I used WHEN_IN_FOCUSED_WINDOW which means that the Grid Panel is registered to receive the action inside the Window which is focused. Then the problem of focus was solved. This part of the script can be found in the class PlayerKeyEventController.java.

Improvements

This project could have no end, many improvements can be done by adding new features. Due to the time available for the project, I was unable to add these features. Here is an exhaustive list of improvement that could be added to the game.

- Change the color of the cells based on the number of iterations of the game
- Thanks to the amazing properties of the game of life, which is the creation of patterns, one could imagine adding enemies, they could move thanks to the different behaviors of the motifs. Thus, if the player can't avoid the enemies and gets hit then the game will be over.
- Adding a timer could add more difficulty of the game.
- Changing the size of the cells during the game or change the size of the grid.

- Fill the grid by reading an external file

- Back to my original idea, changing the dimension of the game to a 3D grid. In fact, I was thinking of applying the grid on a 3D mesh, thus depending on the player's level, we could change the mesh to make the game harder. It would be like flying in a new universe at each level to conquer planets by avoiding the meteoroids.

Documentation

To comment my code, I've used Doxygen. As explain on the Doxygen website (Doxygen, s.d.), Doxygen is the standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL and to some extent D.

Doxygen can help you in three ways:

1. It can generate an online documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating an output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep it consistent with the source code.
2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various element. It includes dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
3. You can also use doxygen for creating normal documentation (as I did for the doxygen user manual and website).

Then I generated a website regrouping all the documentation concerning the project.

Figure 30 : Game of life Space Version documentation website

To give you an overview on the generated website, here's an example of the Classes page:

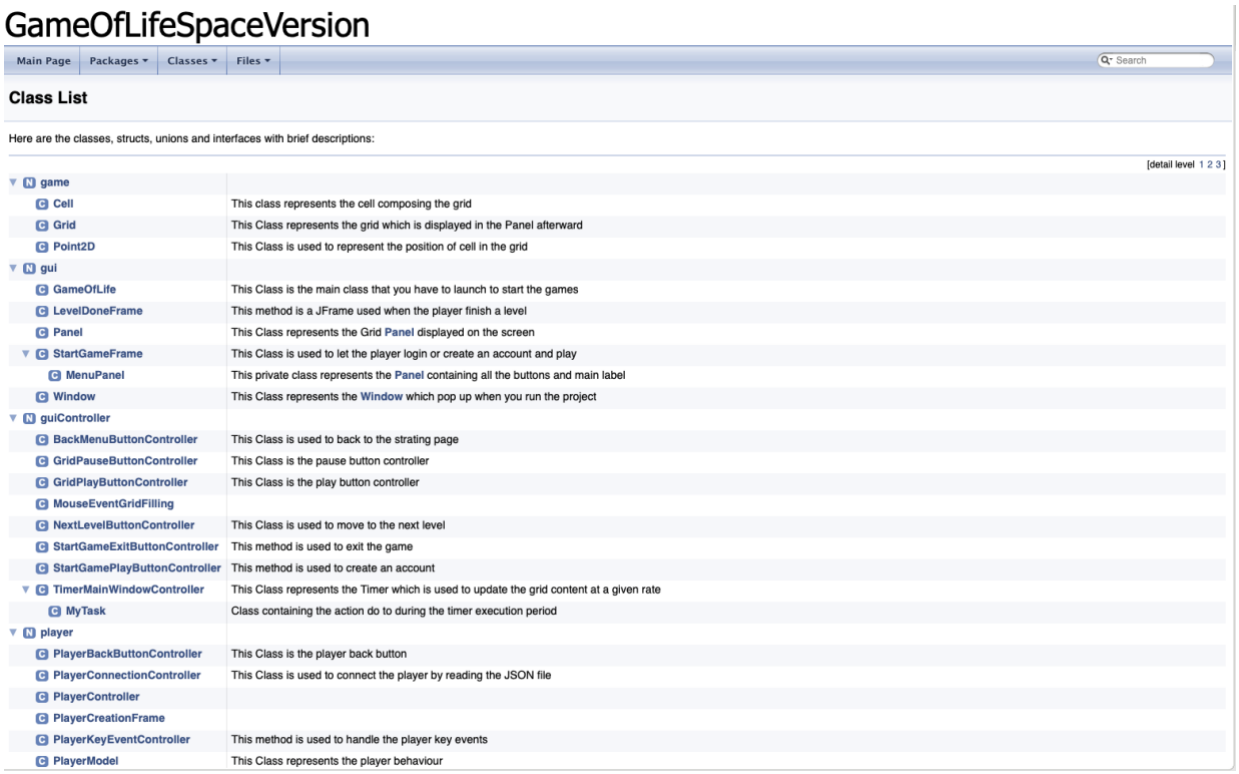


Figure 31 : Doxygen Classes List page

Share

All the resources and explanations about the project are available on GitHub at https://github.com/totordudu/GameOfLife .

Sources

Contributors. (2020, June 2). *Conway's Game of Life*. Retrieved from Wikipedia: https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=960350147

Contributors, W. (2020, May 3). *Model–view–controller*. Retrieved from Wikipedia: https://en.wikipedia.org/w/index.php?title=Model–view–controller&oldid=959941229

Doxygen. (n.d.). *Doxygen*. Retrieved from Doxygen: https://www.doxygen.nl/index.html

Oracle, D. (n.d.). *Class KeyEvent*. Retrieved from Doc Oracle: https://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyEvent.html

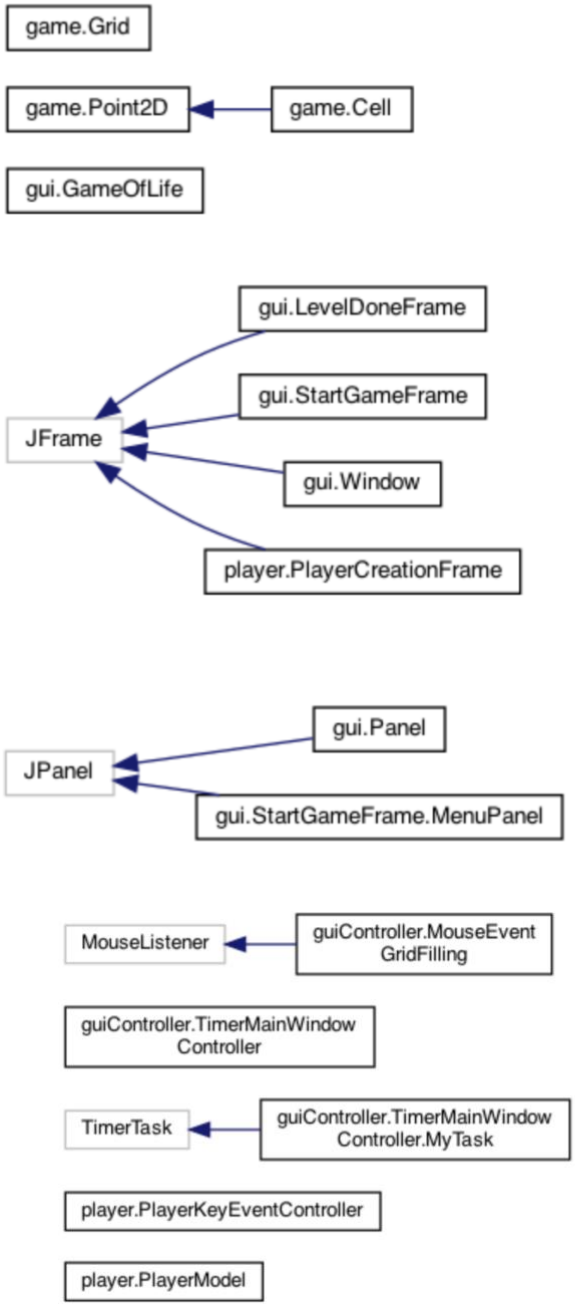
Oracle, D. (n.d.). *How to Use Key Bindings*. Retrieved from Oracle: https://docs.oracle.com/javase/tutorial/uiswing/misc/keybinding.html

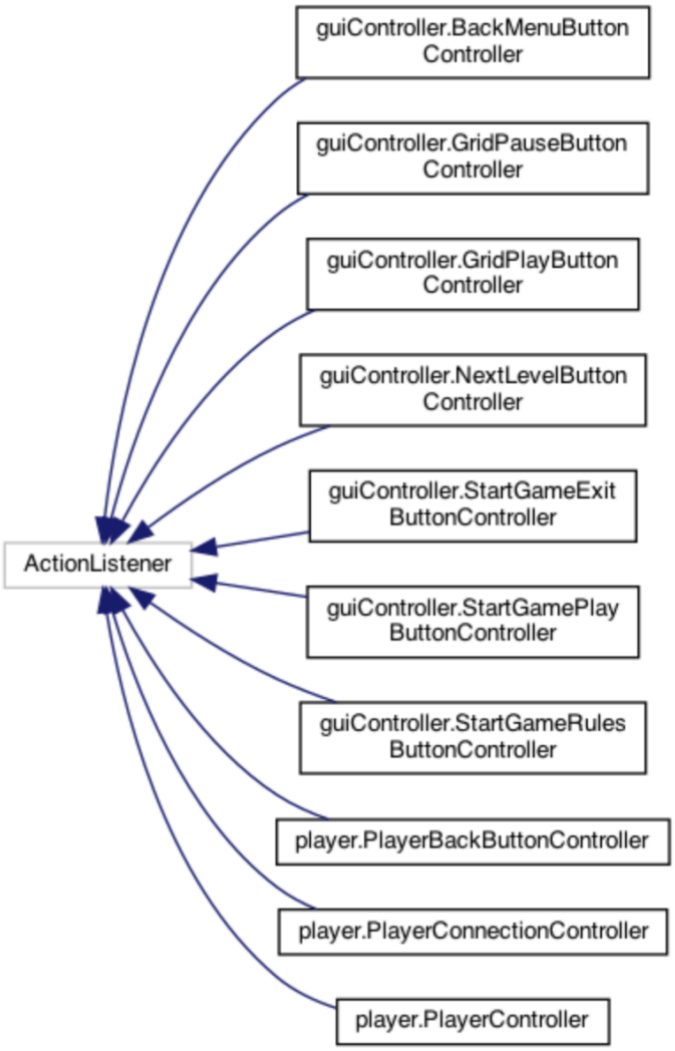
Wolfram. (2020, Juin 3). *Cellular Automaton* . Retrieved from Wolfram: https://mathworld.wolfram.com/CellularAutomaton.html

Wolfram. (2020, Juin 3). *Moore Neighborhood*. Retrieved from Wolfram: https://mathworld.wolfram.com/MooreNeighborhood.html

Annexes

- Classes Organization





As I don’t want to add to much annexes, you can find all the dependencies and exchanges between classes in the images/graphs folder.