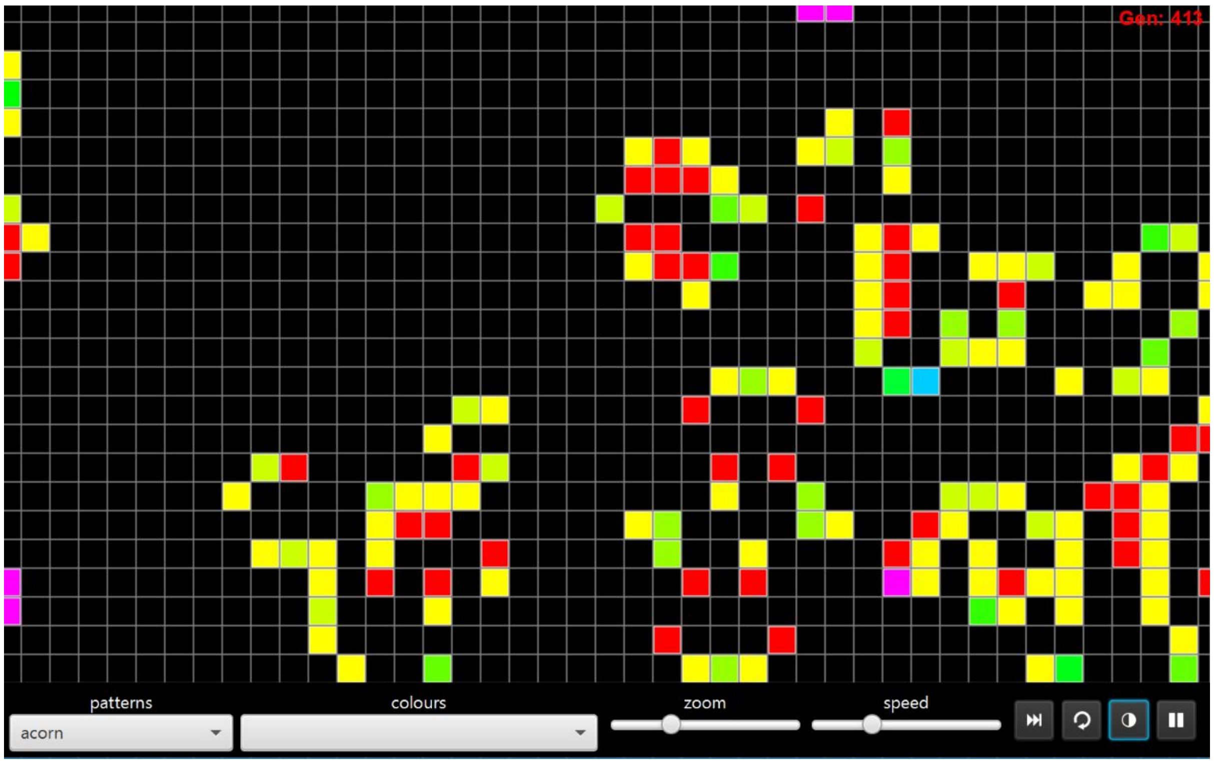
Game of Life Group Project Report

Members:

Bevan, Callum, Guy, Heba, Kelly



Introduction:

Conway's Game of Life is not so much a game, but a simulation. It consists of an infinite grid of cells, where each cell can either be 'alive' or 'dead'. The simulation begins with a number of live cells, and each frame of the animation calculates which cells should die, remain, or be born. This calculation is based on the following rules as described by Martin (n.d.):

- ❖ Cells with one or less neighbours die from underpopulation.
- ❖ Cells with four or more neighbours die from overpopulation.
- ❖ Cells with two or three neighbours survive.
- ❖ Empty cells become alive when they have three alive neighbours.

Our task over four days was to design, create, and present a working Game of Life application. The following report discusses how we organised ourselves as a group, our design, and reflections on the process.

Organisation of the group:

- Whiteboard sessions:
  Brainstorming/visualising, group meetings, assigning/prioritising tasks.

- Individual programming:
  We did this very rarely and mostly when the task was of interest to an individual or required minimal coding. Even in these cases other group members would often check-in.

- Pair programming:
  This was how we worked on specific tasks in groups most of the time.

- Cluster programming:
  We coined this term to describe a style of collaborative work whereby members of a group work together in close proximity, and move fluidly between working individually to working together in pairs or small groups when required for a particular bug/task, and then leave again once that task was resolved. We found this to be more efficient than strictly pair or mob programming at times, as having multiple minds working on one problem was only needed while debugging or solving particularly difficult problems. It allowed us to organically move around to help each other when needed. This was actively occurring at all times during the project.

- Mob programming:
  Occasionally happened when the cluster involved all group members.

- Sprints:
  This approach was used when we were working on tasks that were not top priority and were taking longer than desired. We would set a timer to complete the task (e.g. for 20 minutes) after which if the task was incomplete we would leave it and come back later once other tasks that were more important were complete.

- Slack channel:
  For after-hours communication.

Testing:

Testing was done by all group members. We each made sure the code was working on our individual machines as expected before pushing to git.Testing involved inserting lines in the code and printing them to the console, as well as observing the graphical user interface.

Version control:

After merging our code with the master we would retest to ensure that all our code and any new code was working as expected. Often we experienced merge conflicts and all learned how to use the merge tool effectively. We became more aware of how to avoid merge conflicts by communicating with each other about when to merge; thereby avoiding duplication of effort where multiple people were independently resolving the same merge conflicts. We still have a lot to learn about how to use the branches functionality in git to have a more structured workflow and more effectively avoid merge conflicts.

Roles of each group member:

Roles were decided upon during meetings to identify tasks that needed to be done. We would gather to discuss the direction we were planning to take and formulate a list of tasks to be prioritised. Through this process we understood what individual group members were interested in implementing and what everyone's strengths were. We decided to pair program where possible, making sure the people with strong skills in a particular area took the lead but also that other group members who wanted to learn had the opportunity to work alongside them.

- Kelly: implemented UI design layout and style, pseudo-code and method stubs for the initial design foundation, collaboratively coded game logic, assisted with class diagram, sequence diagram, Javadoc documentation.

- Bevan: general disruptor/scrum master, collaboratively coded game logic, methods for displaying logic of game through colours of cells, colour combo box, UI design, Javadoc documentation, class diagram, assisted with sequence diagram.

- Heba: collaboratively coded game logic, implemented generation counter method to display generations, Javadoc documentation, UI design.

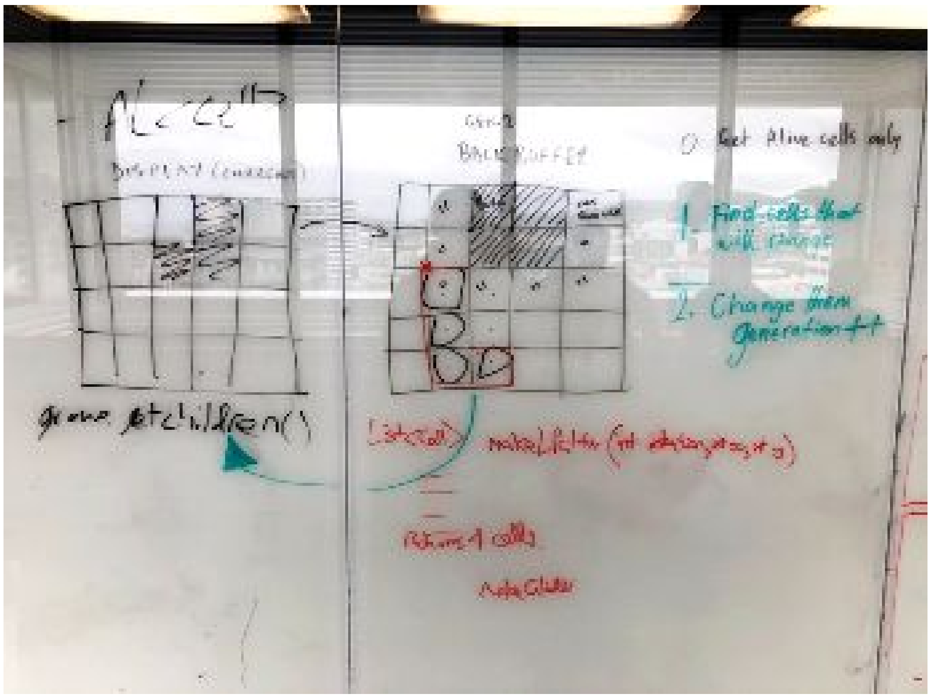
- Guy: core game logic, initial design foundation, grid logic, initial UI layout, implementation of infinite scrolling, speed and zoom sliders, assisted with colours combo box, prototyped pattern parser with grid coordinates for patterns, helping team members understand concepts, UI design.

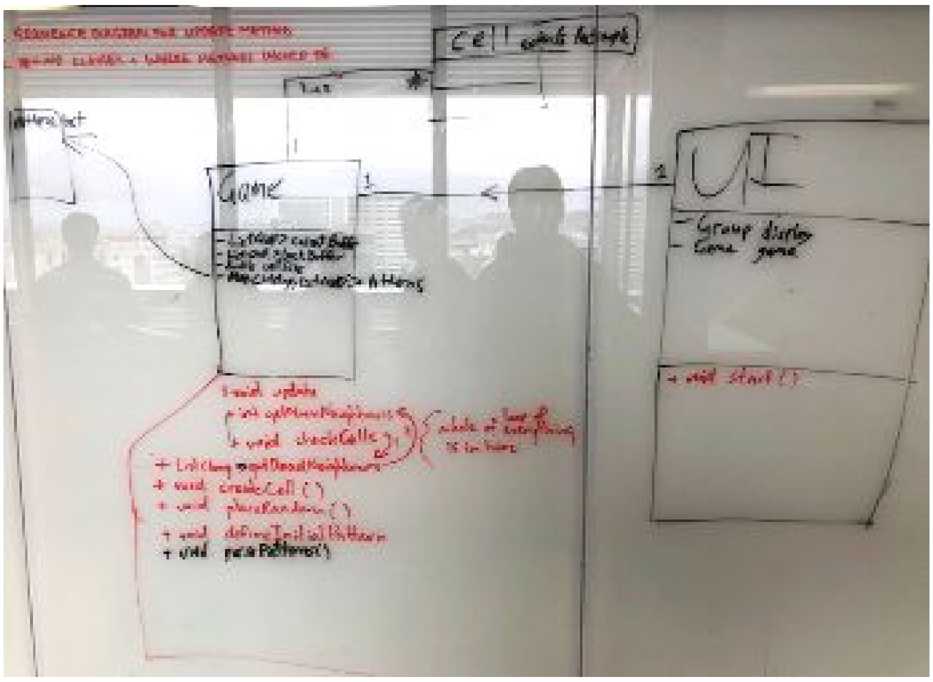
- Callum: core game logic, improved the file to be parsed, researching and incorporating patterns, data structure, UI placing patterns via mouse, layout and design collaboration, combo box for patterns, helping team members understand concepts.

- All group members: debugging and testing, resolving merge conflicts, participating/leading in group discussions, preparing/delivering presentation and group report.

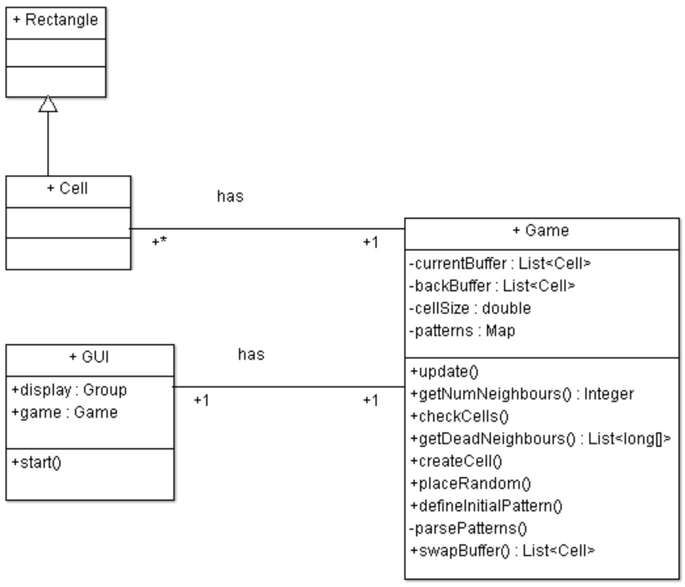
Phases of work:

On Monday we planned both the structure of our project and how we were planning to move forward as a team. We started by independently reading the brief and writing down some initial ideas for the first half an hour. We then brought all of our ideas together and started brainstorming ideas and thinking about different strategies we could take to implement Conway's Game of Life. This led to further discussion on how to go about implementing specific aspects in an efficient manner.



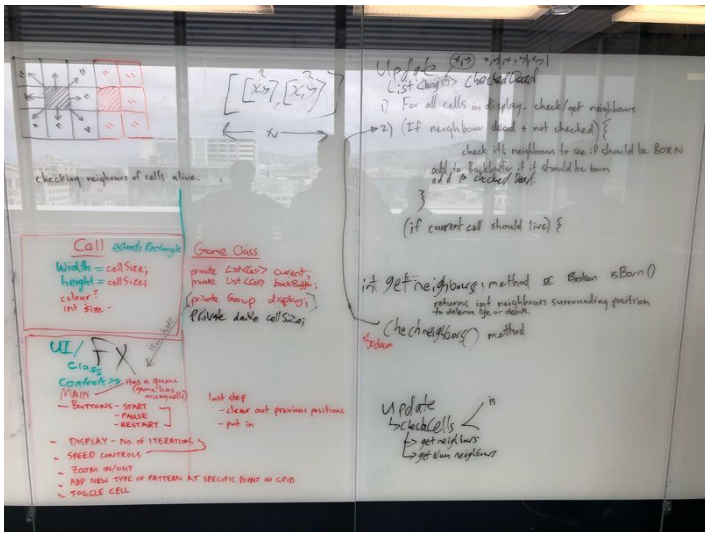


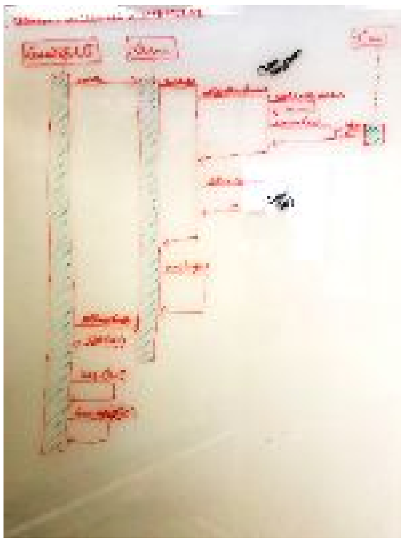
After our brainstorm we started developing a basic class diagram (see Appendix). First we started thinking about the different classes we might need and what sort of data structures we would use to implement our design.



We also started outlining what core methods we would need and what these methods would return. We found a sequence diagram (see Appendix) helpful for interactions between different methods, specifically detailing the process of updating the game with a new generation of cells. This allowed us to consider possible issues that might arise from our planning thus far (be it due to limits of specific data structures or difficulty in designing clear and specific methods, etc) and ultimately we came up with a strong design that at its core didn’t change much if at all from concept to implementation.

Eventually we decided we were unlikely to consider any more potential problems with the design we had come up with without starting to implement it. So we set up our project on GitLab, made sure everyone in the group had access, and began coding at this point.





Design and Implementation:

On Tuesday and Wednesday, we held group meetings to identify and assign tasks to specific group members, discuss design ideas, and how to implement them.

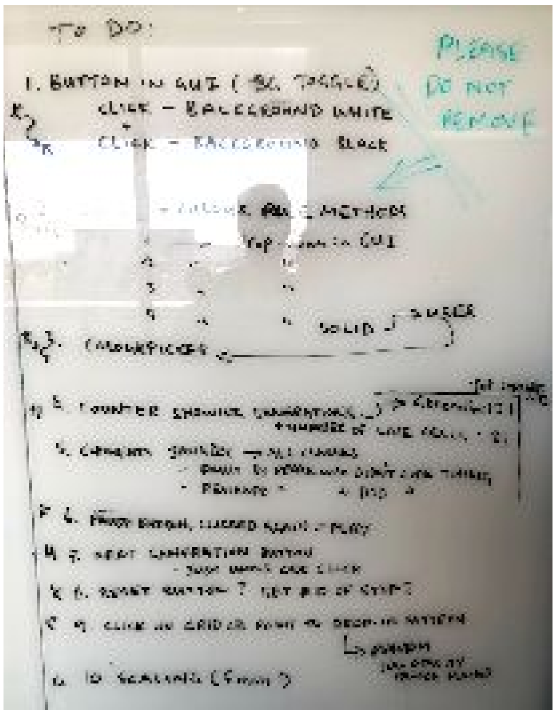
The main aspect we needed to nail down, upon which the rest of the design would rely, was the infinite nature of the 2D grid in which the cells would live. This represented a challenge both in terms of how to visually represent “infinity” (see Appendix), as well as how to hold a potentially infinite number of cells programmatically. Depending on what data structure we used to contain the cells, and how we chose to represent the grid (including the underlying coordinate system), the rest of the design could drastically change.

We realised that trying to contain an infinite number of dead cells in the computer’s memory would cause massive problems (if it was even possible). However the rules that Conway put forth for how the game works provided the solution: dead cells will only come to life if they are in the immediate vicinity of living cells. This meant that so long as we knew the position of the currently living cells, as well as how big each cell is (or rather how far apart each cell’s coordinates are) we could then compute the position of the neighbouring cells, and check whether that cell is occupied by a living cell or not.

Having decided to only keep track of living cells (initially in an ArrayList), we realised that we would have issues iterating over that list and updating it with the next generation. We decided that a good solution would be to use a buffer system, like that used in computer graphics workflows; that is, we would use the information in our current generation (which we chose to call *currentBuffer*) to compute which cells would stay alive in the next generation. That next generation would populate another list called *backBuffer*. Once all cells in the *currentBuffer* have been checked and our *backBuffer* contains our entire next generation, the lists are swapped, and the cells in *currentBuffer* (i.e. the now-current generation of cells) are ready to be displayed.

We now had a good idea of the process of how the *Game* class would implement the rules of the game to generate each generation and prepare them for display. With only living cells to worry about, and a coordinate system wherein each cell’s identifying coordinate was exactly one *cellSize* away from its neighbours in both the x and y axis, the foundation was laid for the implementation of an infinite 2D grid. So long as our visible grid lines continued to move across the screen, it would be obvious that the user was scrolling along even if no cells were at those coordinates on the screen. Because our *Cell* class extends the *JavaFX Rectangle* class, the *Cell* objects could be treated in the same way as *Nodes*. This allowed us to simply translate the *Group* that held the current generation of cells, which doesn’t cause coordinate conflicts as translate values for *JavaFX Node*s are relative to their parent (meaning the cells would translate appropriately along with the *Group)*. By translating the

Group this way, combined with the implementation of scrolling the grid lines across the display, we could create the illusion of an infinitely large grid even though the only grid lines existed were the ones displayed on the screen, and the group of cells itself was only as large as the bounds of the cells which were furthest from one another. The grid lines themselves then, held in a new class *GridBackground*, are built each time the user scrolls along, to ensure that it doesn't try to draw more lines than are visible on the screen at any given point.



TO DO:

PLEASE DO NOT REMOVE

1. BUTTON IN GUI (BG. TOGGLE)
CLICK – BACKGROUND WHITE
CLICK – BACKGROUND BLACK

[illegible] METHODS
[illegible] IN GUI

SOLID
SLIDER

3. COLOURPICKER

5. COUNTER SHOWING GENERATIONS [illegible]
NUMBER OF LIVE CELLS [illegible]

6. [illegible]

6. [illegible] BUTTON, CLICKED AGAIN = PLAY

7. NEXT GENERATION BUTTON
[illegible]

8. RESET BUTTON ? [illegible]

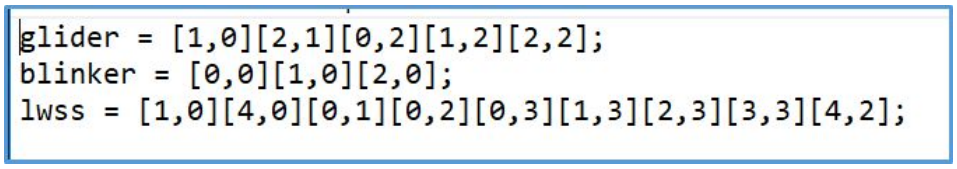
9. CLICK ON GRID [illegible] PATTERN

10. SCALING (ZOOM)

After implementing the core mechanics of the program (namely the implementation of Conway's rules), we had a group meeting about additional functionality and features that we had identified in our initial brainstorming session. We marked down who would be primarily responsible for which elements and went back to work. Those elements were:

- ❖ the ability to select and place predefined patterns of cells at specific coordinates on the grid;
- ❖ various rulesets to define the colours of each cell (e.g. basing colours off how many generations each cell has been alive or how many neighbours each cell has, etc); and
- ❖ additional UI elements such as toggling the background between black and white, a speed slider, and a zoom slider.

In order to select predefined patterns, we realised we would need to store the templates for those patterns somewhere. We considered having a class to store that information, but ultimately decided that having a non-Java file (which we gave the extension *gol*, short for Game of Life) would allow us more flexibility, allowing the future possibility of user-supplied pattern files. We created a *PatternParser* class which scanned through the file, finding the name of each pattern and the coordinates at which each cell in the pattern would lie relative to the top-left cell of the pattern (which would be translated depending on the position of the user's mouse on the grid). Initially the file format listed discrete coordinates of cells (see image below), however we realised that for larger patterns it was laborious to calculate all the coordinates we required. It was decided that a more visual approach would be appropriate as detailed in the reflection section.



```
glider = [1,0][2,1][0,2][1,2][2,2];
blinker = [0,0][1,0][2,0];
lwss = [1,0][4,0][0,1][0,2][0,3][1,3][2,3][3,3][4,2];
```

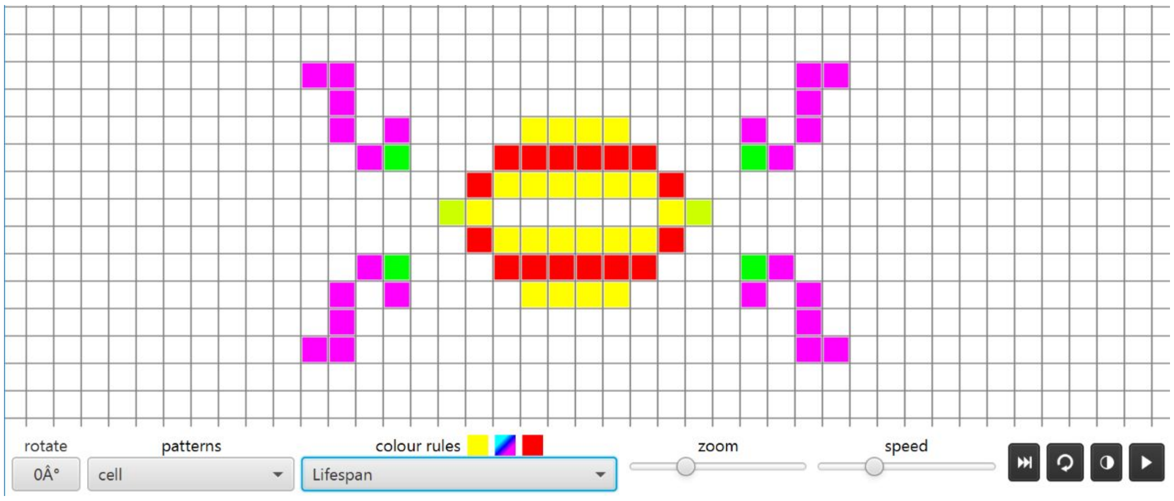
❖ Above: Initial file format which listed cell coordinates.

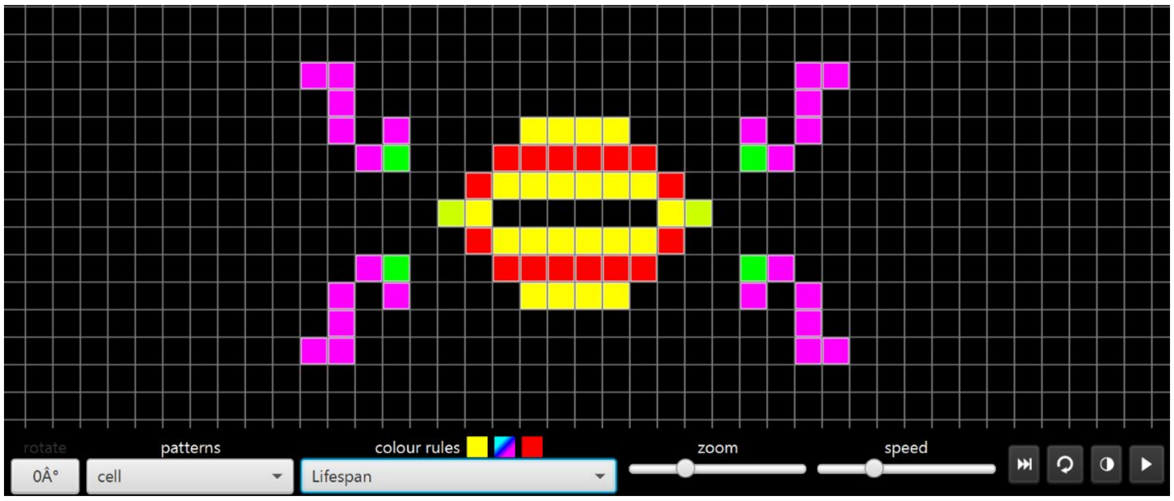
Then we created various rulesets to define the colours of each cell. In addition to the user being able to use a *ColorPicker* to choose their "custom" colour of choice for the display of the cells, we thought it would be useful (and fun!) to have methods that would display the rules of Conway's Game of Life through colour.

The method *colorRuleNeighbours()* assigns colours depending on the amount of neighbours a cell has (e.g., 1 neighbour = yellow, 2 neighbours = orange, and so on up to the maximum number of 8 neighbours). This means that most of the time in the display you see colours that relate to cells having between one and four neighbours, with rarer colours (5 or more neighbours) appearing sporadically.

The method *colorRuleLifeSpan()* more directly shows the logic of Conway's Game of Life. Cells that are born appear yellow, and cells that will die in the next generation appear red (unless they appear for only one generation in which case they are yellow). All other live cells slowly transition in colour until they eventually stabilise and become purple.

We also included the *colorRuleRandom()* method for visual comparison between Conway's rules and a random placement of colours. The only colours that *colorRuleRandom()* does not allow are black and white, as these are our background colours. We wanted to have the ability to switch between a black and white background to allow the user to have contrast options (i.e., light colours are easier to see against a dark background and vice versa).


rotate: 0Â°
patterns: cell
colour rules: Lifespan


rotate: 0Â°
patterns: cell
colour rules: Lifespan

- The above images show the comparison between a light background (above) and a dark background (below).

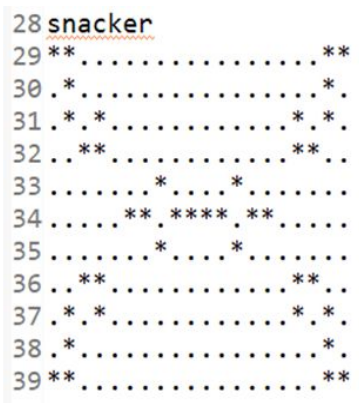
Reflection:

On Tuesday we changed the *currentBuffer* and *backBuffer* from being a *List* to a *Map*, where the key was the position represented as a *double[]*, and the value is the *Cell* object. This was done to allow faster computation of getting neighbours; instead of iterating over the entire list for each cell for which we need the neighbours, we can now just ask the map what (if anything) is at a specific position.

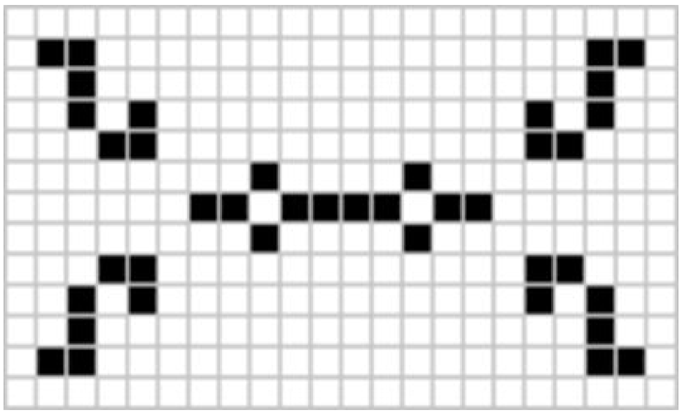
Midday Tuesday we found that when using *currentBuffer.contains(key)* or *currentBuffer.get(key)* using a key value of *double[]*, it wasn't returning *true* (or the object, in the case of *get()*) when the two keys should theoretically match. The hypothesis was that the *double[]* key wasn't properly hashable, so when two were compared, their respective hashcodes weren't matching. With further research we found that our theory was correct; the *Array* class' *hashCode()* method is based on the object's position in memory, rather than the contents of the array. Our solution was to create a new *Position* class that holds two double values (x and y), and overrides the *hashCode()* method with a method generated automatically by Eclipse, which can return the coordinate values as an array or as individual values. As a result, we changed the *currentBuffer* and *backBuffer* key values to type *Position*, as well as refactoring the *getDeadNeighbours()* and *getNumNeighbours()* methods. This worked, and had the added benefit of being more concise and easier to read (where previously you would have to guess that *double[]* represented a cell's position, the class name *Position* relays that intent instantly).

We had several issues with zooming/scaling, specifically with the grid drawn by the *GridBackground* class; initially we were attempting to scale the grid separately from the game's cell contents. The issue with this was that the grid scaled from the top left corner of the window, while the game contents would scale from the centre of the game group's bounds. Attempting to counteract the difference in scale centre positions proved more difficult than it was worth. Ultimately we found that placing both the grid and the game contents into an intermediary scale offset group allowed us to centralise them both with the same consistent scaling centre. We then had to ensure that the grid was always big enough by default so that when you zoom out, the grid stretches to the edges of the window.

During the initial brainstorming session on Monday, we had the idea to create a *PatternParser* class to make entering different cell patterns easier, and allowing this pattern setup to be separated out into a text file. This worked well, however it required figuring out and entering the coordinates of each cell in a pattern, which in the future would become a problem when entering myriad patterns, many of which can be upwards of 40 x 20 cells. The solution to this was altering the parser so that it could read * and . symbols (representing live cells and empty spaces respectively) in a grid layout, which proved much faster to manually enter into the text files. Serendipitously, we found a website collated by Hensel (1995) that lists dozens of Game of Life patterns that have been created over the years in this format already, so they could be copied and pasted into the text file with minimal effort.



```
snacker
**................**
.*................*.
.*.*............*.*.
..**............**..
.......*....*.......
.....**.****.**.....
.......*....*.......
..**............**..
.*.*............*.*.
.*................*.
**................**
```



❖ The above images show the text file (left) and the cell display (right) for a “snacker”.

One feature that proved particularly challenging was having the chosen pattern follow the mouse while choosing where to place it on the grid. This was Callum’s main goal to implement on Wednesday. After four hours the core implementation was working, however the pattern following the mouse would delete cells that were already placed on the grid. Pair programming took over, and the way the feature was currently working and the bug were explained to Guy, who realised that the temporary pattern should be added to the *displayBuffer* directly, rather than entering into the *Game* class’ computational buffers. This fixed that issue, but shortly thereafter we found a different problem wherein the tracking of the pattern to the mouse didn’t line up when the grid was zoomed out or scrolled away from the origin (that is, the pattern’s coordinates in the grid were not at the position of the mouse on the screen). Many attempts were made to fix this, using the *displayBuffer* translate properties (which were being used to offset the pattern placement from the group’s translation due to scrolling the grid) and the current level of zoom to alter the position of the pattern relative to the mouse coordinates. Bevan then had a quick look at the problem and suggested that the scale should affect the mouse coordinates only, rather than adjusting the grid’s translation offset. This was the perfect solution, and it was the genesis of the collaborative programming style we’ve coined ‘cluster programming’.

Implementing the UI side of the colour rulesets proved to be a challenge. We knew that having simple strings representing the rules in our *ComboBox* would work, but we wanted to also display the specific colours that each ruleset used within the drop down menu itself. Initially we thought that simply setting up each ruleset as a *Text* object and a series of *Rectangles* inside an *HBox* would work, and the *HBox*es could then be added to the *ComboBox*. This appeared to work initially, until an item was selected, at which stage it would appear on the *ComboBox* button but disappear from the drop down list. We discovered that this was because under the hood *ComboBox* uses a *Button* and a *TreeView*, which are distinct nodes in the stage hierarchy. When selecting an option, it adds the selection to the *Button*, but because nodes can only be in one place in a stage hierarchy, it would no longer exist in the *TreeView*.

We found that this was a well documented problem, and that the suggested solution was to use a *CellFactory*, which defines the manner in which each cell of the *TreeView* is constructed and how it responds to being selected. This solved the issue of each item disappearing when selected. But this presented a new issue which was that when selected, the items were represented in the button as a string of the object (i.e calling the *toString()* method on the object) held by the *CellFactory*. Because we needed to represent both text and colours in the cells, that object was a map containing a *String* (the name of the colour rule) and a *Color* array (to represent the colours that are used in that rule). The resulting string that displayed on the *ComboBox* button, therefore, was essentially garbage. We tinkered with it a bit more and were able to have only the colours display in the button, but they wouldn't be replaced by the next selection, appending on to a never-ending line of colours which expanded beyond the bounds of the button itself. While we were working on this, others in the team were working on merge conflicts. We decided that this was only a visual issue and not essential to the functioning of the product, so if we did not find a solution before the conflicts we resolved, we would move on to implementing the actual rules themselves.

Ultimately we were able to resolve the issue with some help from Craig; he suggested that we could have a label above the *ComboBox* that held the name of the currently selected item, and the colour next to it. We adapted the suggestion, having the label describe the purpose of the *ComboBox* instead, as we found that a *StringConverter* would allow us to convert the Map into a meaningful string (i.e. using the key, which was the name of the rule) and show that in the button, and have the colours of the selected rule displayed above the button.

Improvements

Features we would have liked to add to the project:

- ❖ A panel in the GUI to display a colour legend with labels alongside coloured squares to explain to the user what each colour means for the option they have selected.

- ❖ A hotkey for rotating a selected pattern 90 degrees in one direction any number of times before it is placed onto the screen (similar to the rotate key in the classic computer game *Tetris*).

- ❖ A class to act as an intermediary between the *Game* class and the *GameOfLifeUI* class. This class would take the window dimensions, scale, and scroll/translation values into account, decide which cells from the *currentBuffer* should be visible, where to place them in the window and how large, and pass that information to the UI to be displayed. It would have provided a cleaner solution to some of the issues we faced, made the project more scalable, and also resolved an issue we only recently noticed wherein if a cell moves beyond the bounds of the *scaleOffset Group*, that group moves around on its own to compensate.

Technical aspects of working in a group we would like to improve on:

❖ More organisation on who and when to merge. Utilising branches instead of everyone committing to the master branch.

References

Martin, E. (n.d.). *John Conway's Game of Life*. Retrieved August 21, 2018, from https://bitstorm.org/gameoflife

Hensel, A. (1995). *A Brief Illustrated Glossary of Terms in Conway's Game of Life.* Retrieved August 22, 2018, from http://www.radicaleye.com/lifepage/glossary.html

Appendix

Representing Infinity

It is important to note that computers can only hold a finite amount of information, and that in Java there are predetermined minimum and maximum values that primitive types can represent. The idea of an "infinite" coordinate system therefore really is an illusion that would potentially break if either the number of cells approached the limit of memory a computer could store, or their positions approached the minimum or maximum values that could be represented by the underlying primitives.

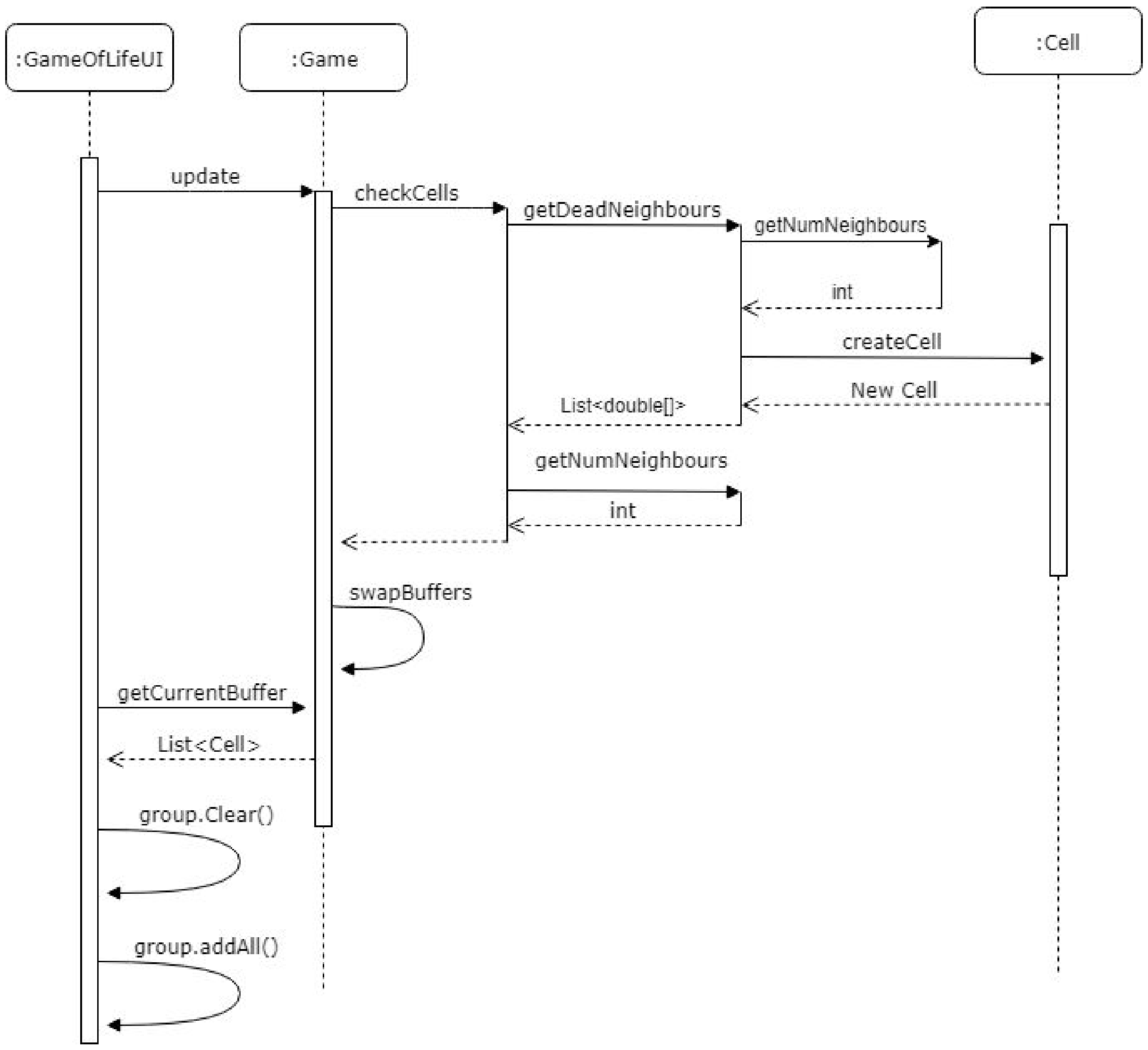
In our case, we used doubles to represent our positions; if a cell's position exceeded Double.MAX_VALUE or Double.MIN_VALUE, it would fall back to that respective value. For our implementation, that would mean that *Game* would think those cells are their own neighbours, which would violate Conway's rules. This in turn could potentially override those cells in the map, instead of generating new cells when the rules dictate new cells should be generated.

One possible solution would be that when a cell approaches the min or max value represented as a double, it is told that its neighbours are at min or max values respectively, resulting in an infinitely looping grid rather than an infinitely expanding one. Using a long instead of a double in the first place would automatically result in this behaviour, as Long.MAX_VALUE +1 = Long.MIN_VALUE, and Long.MIN_VALUE -1 = Long.MAX_VALUE. It might be prudent, therefore, for us to use a long to represent Position coordinates. This was not taken into consideration until after the project was complete, and would probably be a change we would enact if we were to continue with this project in the future.

Regarding the potential for the number of cells to exceed that which the computer could reliably contain in memory, it might be wise to implement some limits. For example, if the number of cells being tracked exceeds some appropriate number, cells that are significantly distant from the coordinates being viewed by the user could be removed, as it is highly improbable the user would find them by scrolling (considering that the area the user can scroll to is essentially infinite).

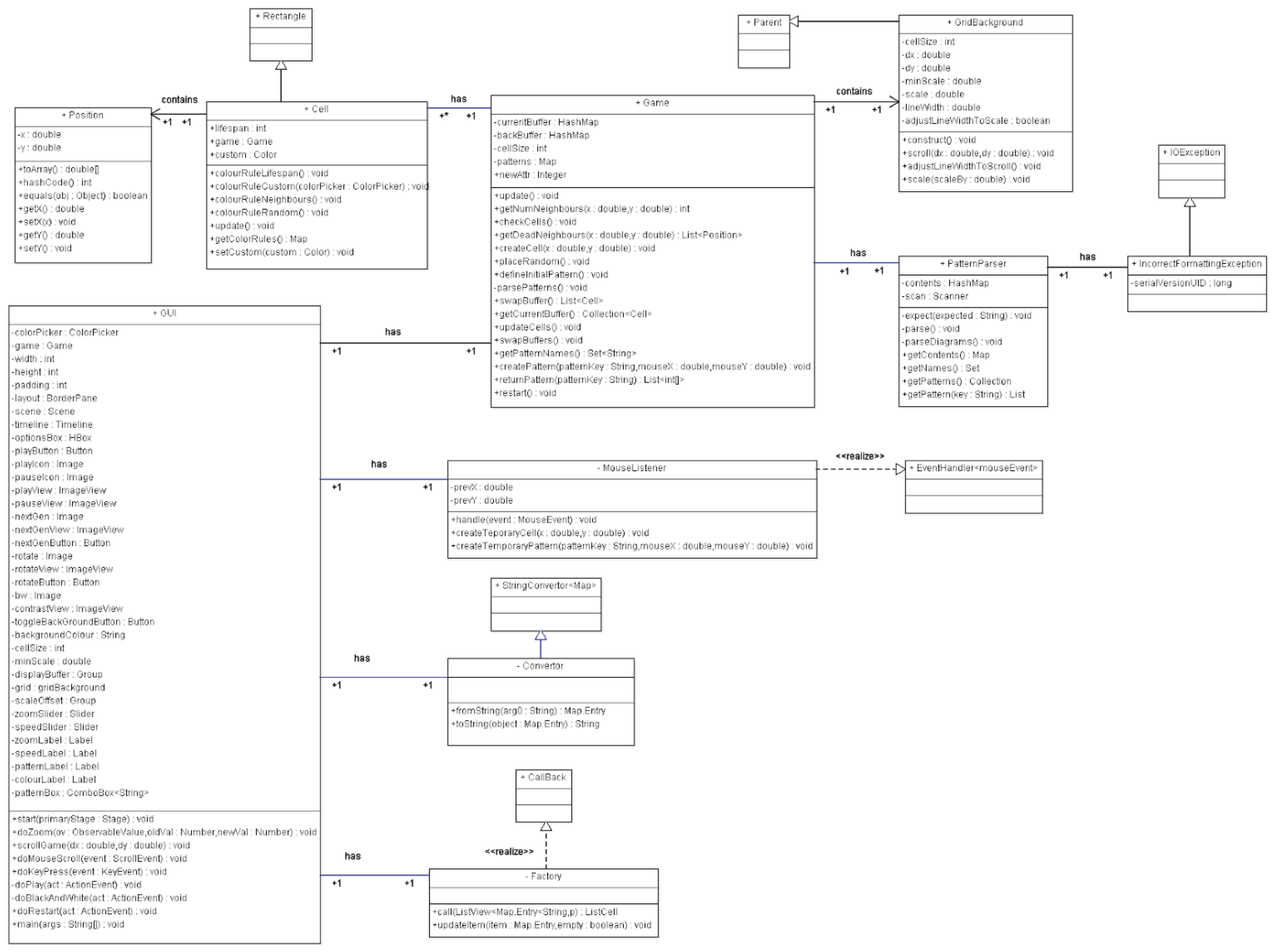
Sequence Diagram

A more readable copy of our sequence diagram follows. It details the sequence that takes place each frame when the game is playing in order to apply Conway's rules, create and retain cells as appropriate, and collect those cells from the *Game* class to be displayed by the *GameOfLifeUI* class.

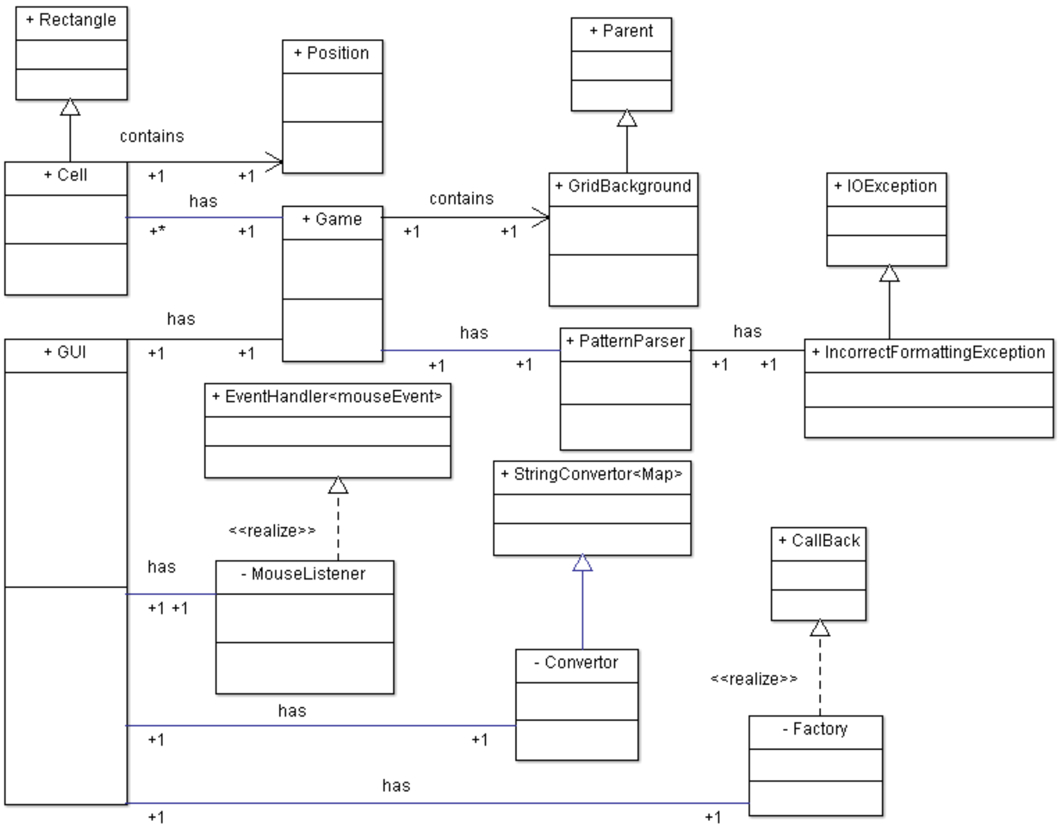


Class Diagram

This class diagram shows all classes and their methods at the end of our development phase. A simplified version has also been included for readability. While there is added complexity in comparison to our original design, the core design remains unchanged.



+ Rectangle
+ Parent
+ GridBackground
-cellSize : int
-dx : double
-dy : double
-minScale : double
-scale : double
-lineWidth : double
-adjustLineWidthToScale : boolean
+construct() : void
+scroll(dx : double,dy : double) : void
+adjustLineWidthToScroll() : void
+scale(scaleBy : double) : void
contains
+ Position
-x : double
-y : double
+toArray() : double[]
+hashCode() : int
+equals(obj : Object) : boolean
+getX() : double
+setX(x) : void
+getY() : double
+setY() : void
+ Cell
+lifespan : int
+game : Game
+custom : Color
+colourRuleLifespan() : void
+colourRuleCustom(colorPicker : ColorPicker) : void
+colourRuleNeighbours() : void
+colourRuleRandom() : void
+update() : void
+getColorRules() : Map
+setCustom(custom : Color) : void
has
+*
+1
+ Game
-currentBuffer : HashMap
-backBuffer : HashMap
-cellSize : int
-patterns : Map
+newAttr : Integer
+update() : void
+getNumNeighbours(x : double,y : double) : int
+checkCells() : void
+getDeadNeighbours(x : double,y : double) : List<Position>
+createCell(x : double,y : double) : void
+placeRandom() : void
+defineInitialPattern() : void
-parsePatterns() : void
+swapBuffer() : List<Cell>
+getCurrentBuffer() : Collection<Cell>
+updateCells() : void
+swapBuffers() : void
+getPatternNames() : Set<String>
+createPattern(patternKey : String,mouseX : double,mouseY : double) : void
+returnPattern(patternKey : String) : List<int[]>
+restart() : void
+ IOException
+ PatternParser
-contents : HashMap
-scan : Scanner
-expect(expected : String) : void
-parse() : void
-parseDiagrams() : void
+getContents() : Map
+getNames() : Set
+getPatterns() : Collection
+getPattern(key : String) : List
+ IncorrectFormattingException
-serialVersionUID : long
+ GUI
-colorPicker : ColorPicker
-game : Game
-width : int
-height : int
-padding : int
-layout : BorderPane
-scene : Scene
-timeline : Timeline
-optionsBox : HBox
-playButton : Button
-playIcon : Image
-pauseIcon : Image
-playView : ImageView
-pauseView : ImageView
-nextGen : Image
-nextGenView : ImageView
-nextGenButton : Button
-rotate : Image
-rotateView : ImageView
-rotateButton : Button
-bw : Image
-contrastView : ImageView
-toggleBackGroundButton : Button
-backgroundColour : String
-cellSize : int
-minScale : double
-displayBuffer : Group
-grid : gridBackground
-scaleOffset : Group
-zoomSlider : Slider
-speedSlider : Slider
-zoomLabel : Label
-speedLabel : Label
-patternLabel : Label
-colourLabel : Label
-patternBox : ComboBox<String>
+start(primaryStage : Stage) : void
+doZoom(ov : ObservableValue,oldVal : Number,newVal : Number) : void
+scrollGame(dx : double,dy : double) : void
+doMouseScroll(event : ScrollEvent) : void
+doKeyPress(event : KeyEvent) : void
-doPlay(act : ActionEvent) : void
-doBlackAndWhite(act : ActionEvent) : void
+doRestart(act : ActionEvent) : void
+main(args : String[]) : void
<<realize>>
- MouseListener
-prevX : double
-prevY : double
+handle(event : MouseEvent) : void
+createTeporaryCell(x : double,y : double) : void
+createTemporaryPattern(patternKey : String,mouseX : double,mouseY : double) : void
+ EventHandler<mouseEvent>
+ StringConvertor<Map>
- Convertor
+fromString(arg0 : String) : Map.Entry
+toString(object : Map.Entry) : String
+ CallBack
- Factory
+call(ListView<Map.Entry<String,p) : ListCell
+updateItem(item : Map.Entry,empty : boolean) : void



+ Rectangle
+ Position
+ Parent
contains
+ Cell
+1
+1
has
+ Game
+*
+1
contains
+ GridBackground
+1
+1
+ IOException
has
+ GUI
+1
+1
has
+ PatternParser
+1
+1
has
+ IncorrectFormattingException
+1
+1
+ EventHandler<mouseEvent>
+ StringConvertor<Map>
<<realize>>
+ CallBack
has
- MouseListener
+1 +1
- Convertor
<<realize>>
has
- Factory
+1
+1
has
+1
+1