UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

2° Cuatrimestre de 2022

(95.15) Algoritmos y Programación II

Trabajo Práctico

Juego de la Vida V1.0

Alumnos	Padrón	Mail
Seivane, Andrés Francisco	98810	aseivane@fi.uba.ar

Índice

1. Introducción

En el presente trabajo se realizó un programa que permita juegar al “juego de la vida". En este se debe introducir la ubicación de las células iniciales y decidir paso a paso si se quiere jugar el próximo turno, reiniciar el juego o terminarlo. El juego de la vida es en realidad un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior.

2. Objetivo

Realizar una aplicación que lleve adelante la ejecución del juego de la vida en un tablero limitado, de 20 filas por 80 columnas, a partir de una configuración inicial de células vivas indicadas por el usuario.

3. Manual del usuario

3.1. Reglas del juego

El “tablero de juego” es una malla formada por celdas (“células”), que está limitada a 20x80. Cada célula tiene hasta 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales. Las células tienen dos estados: están “vivas” o “muertas”.

El estado de la malla evoluciona a lo largo de cada turno. El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente.

Las transiciones dependen del número de células vecinas vivas:

- Una célula muerta con exactamente 3 células vecinas vivas “nace” (al turno siguiente estará viva).
- Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere o permanece muerta (por “soledad” o “superpoblación”).

3.2. Como jugar

- El juego cuenta con un tablero de 20x80, el cual se configura en la etapa inicial y luego no puede ser modificado.
- Cuando el juego inicia, se le dará una bienvenida.
- En primer instancia se debe introducir la cantidad de células vivas iniciales. Se debe ingresar un valor entre 0 y 1600.
- Luego, una a una se solicitará que ingrese el índice de fila y columna donde se encontrarán las células vivas. Siendo la célula ubicada fila 1 y columna 1, la que se encuentra en la esquina superior izquierda. Para las filas debe ser un valor entre 1 y 20. Para las columnas debe ser un valor entre 1 y 80.
- Habiendo introducido todas las células, se muestra en pantalla el tablero de acuerdo a los valores ingresados en los pasos anteriores.
- A partir de este momento el juego no se puede modificar. Sólo se puede avanzar el turno, reiniciar el juego o terminarlo.

4. Manual del programador

4.1. General

La Figura 1 denota el diagrama de "clases". Aunque no se utilizaron clases, se realizó el esquema para representar como están relacionados los objetos del programa. Este programa no valida datos del usuario. La validación de datos es escencial pero no estuvo en el scope del presente trabajo.

Para el programa se buscó un patron tipo MVC (Model-View-Controller). *StateMachine* funciona como un controlador que interactúa con el modelo, el cual está representado por *Game* y la interacción con el usuario se realiza en *Interface*.

El programa se desglosa de una manera intuitiva. El juego *Game* tiene la lógica del juego que opera sobre el tablero Board, el cual a su vez esta compuesto de una grilla de celulas Cell. Para obtener estadísticas como saber cuantas celulas murieron, nacieron, cuantas hay vivas, etc, el juego tiene estadisticas *Statistics*.

Cell es la menor medida del juego. Esta es una unidad que su principal objetivo es saber si esta viva o muerta. Por lo tanto tiene sus setters y getters para obtener y cambiar su estado. La función *getState* casi no se usa, ya que preguntar si la célula esta viva con *isAlive* es más coherente.

Board es un tipo de datos que representa el tablero. Este tendrá una grilla de celulas y los datos sobre sus dimensiones. Este será el encargado de poder decirle al juego el estado de una célula según sus coordenadas, la cantidad de celulas vecinas vivas tiene una determinada, los límites del tablero y, en base a esto, si una posición recibida es válida.

Statistics es un objeto que lleva la cuenta desde el principio de celulas vivas, el turno, la cantidad de nacimientos y muertes de la última partida, y operaciones sobre estos datos. Al tener este objeto por separado es fácil implementar nuevas métricas.

Game tiene en su interior la lógica del juego, por lo que en cada ronda deberá jugar analizando célula por célula cual es su estado y cuales son los pasos a seguir deacuerdo a sus vecinos. Si la célula esta viva, ¿debe seguir viva o debe morir? En caso de que cambie su estado, se lo dirá a Statistics para que lleve la cuenta de los cambios. Game cuenta con dos tableros. Esto se decidió así ya que cada vez que una célula deba mutar no lo puede hacer en el mismo tablero ya que cambiaría las condiciones de las demas. Es por esto que se evalúa en un talero y se anotan los resultados en otro, donde ronda a ronda cambian a donde apuntan los punteros *actualBoard* y *auxBoard*.

Interface es el encargado de mostrar mensajes en pantalla, mostrar el tablero y pedir datos al usuario para que luego StateMachine se lo pase a Game para su posterior procesamiento.

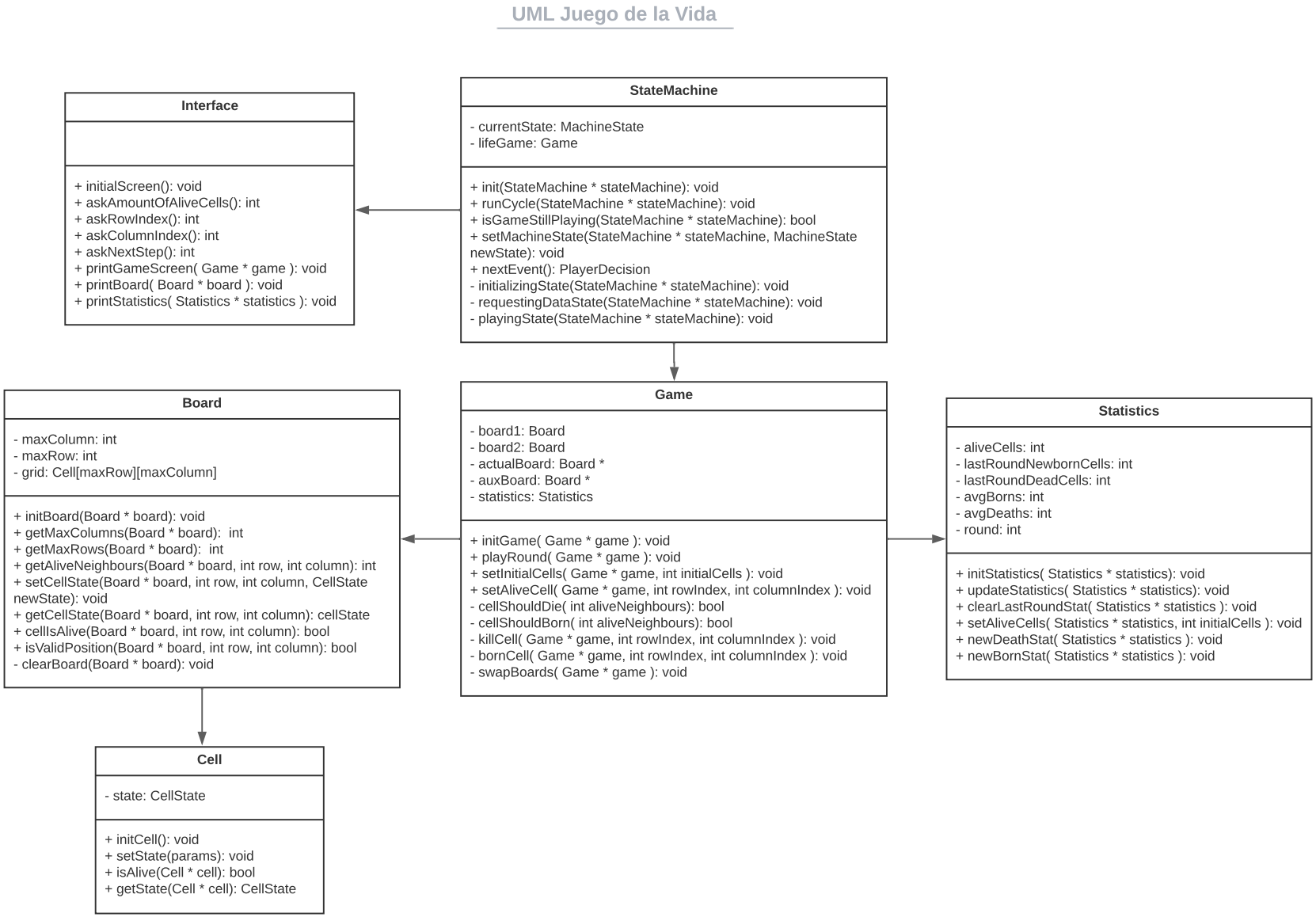

Figura 1: Diagrama UML

4.2. Maquina de estados

El funcionamiento básico del programa es una máquina de estado que tiene definidos los siguientes:

- Initializing
- Requesting Data
- Playing Round
- Ending

En la Figura 2. Se define el flujo que esta seguirá.

Initializing se inicializan las propiedades de Game a sus valores default. Dado que Game contiene otros objetos como *Board* y *Statistics*, estos también se inicializan desde Game.
Requestig Data luego de haberse iniciado el programa, se pedirán las condiciones iniciales del juego. StateMachine le pide a Interface estos datos y se los pasa a Game para que a su vez se lo pase a quien corresponda. Para el caso de la cantidad de células iniciales se las pasa a Statistics. La carga de cada célula se la pasa a Board y este se encarga de asignarlo a la célula correspondiente.
Playing Round es el estado donde se ejecuta la lógica del juego. Game se encargará de pedirle a Board las condiciones que necesite del tablero para definir el estado futuro de cada célula. Cada vez que cambie el estado de alguna, Game se lo hará saber a Statistics. De esta manera, cuando termine el round se actualizarán sus métricas. Luego de cada turno, la maquina de estado le pasa a la interface los datos a imprimir. Este es el único estado que espera un evento externo: la decisión del usuario. Esta puede ser continuar con el juego (continuar en el mismo estado), reiniciarlo (pasar a Initializing) o terminarlo (Ending).
Ending es un estado que no se se implementó nada dentro. Cuando llega a este estado el juego termina. Se podría haber implementado una metrica final, un mensaje de despedida, etc.

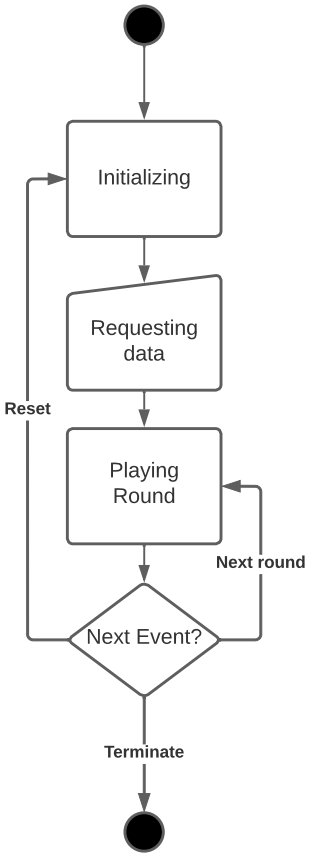

Figura 2: Diagrama maquina de estados

5. Debug

5.1. ¿Qué es Debug?

Debug es un proceso que consta de ejecutar el código paso por paso para identificar donde puede estar un problema, aislándolo y corrigiéndolo.

5.2. ¿Qué es un “Breakpoint”?

El breakpoint es un punto establecido por el usuario donde quiere que el proceso del programa pare ya que se encuentra en una zona critica o cercana a donde está el error. Segun el lenguaje y el IDE, cuando la ejecucion del programa se pausa se puede observar información de la memoria, registros, variables, etc.

5.3. ¿Qué es “Step Into”, “Step Over” y “Step Out”?

Para moverse dentro del código durante la ejecución se puede hacer de un breakpoint a otro o mediante pasos:

- Step Into: permite ejecutar paso por paso y en caso de que sea una función, acceder a ella para ver que ocurre dentro.
- Step Over: funciona de la misma manera que el comando anterior pero en caso de encontrarse con una función esta lo "pasa por encimaz ejecuta la función sin entrar paso por paso a ella.
- Sep Out: se utiliza cuando se accede a una función con Step Into pero no se desea continuar paso a paso dentro de esta. Ejecutando Step Out el IDE ejecuta el resto del código y frena cuando sale de la función.

6. Conclusiones

En el presente trabajo se realizó un programa tratando de mantener las buenas práctica como desarrollar un código ordenado, con nombres representativos, modular y comentado para explicar su proceso. Antes de llevar a cabo del código, se realizo un diagrama UML para entender cuales serían los objetos necesarios y su relación entre si. El código cuenta con todas las especificaciones del apéndice A y se utilizó GitHub[1] para su versionado.

[1] https://github.com/aseivane/Algo2-TP1