Introduzione

Il progetto consiste nella realizzazione del gioco Game Of Life, questo gioco si basa su di una matrice , dove all'interno di ogni cella , possiamo avere o una cellula viva o una cellula morta.
Ad ogni tot di tempo prestabilito , viene visualizzata una nuova matrice dettata dalle regole del ciclo di vita della partita .
Il ciclo di vita prestabilito , consiste che se una cellula è viva :

- muore se ha meno di 2 cellule vive vicine o più di 3

se una cellula è morta :

- vive se ha 3 cellule vive vicine

Ogni partita e' identificata da un Id che corrisponde al nome della partita (univoco) , da una matrice , da un tempo di riposo tra ogni ciclo di vita e dal numero di cicli di vita attuali .

Descrizione del progetto

Il progetto svolto segue la struttura del pattern design MVC , che consiste in una divisione delle classi in base al loro uso (Model , View , Controller), cioè una divisione tra le classi che rappresentano il dato che si andrà ad utilizzare , le classe gestiranno la parte di interfacciamento grafico con l'utente , e le classi che hanno il compito di manipolare i dati che andremo ad utilizzare e di fare a ponte tra il Model e la View.

Model Package

In questo package, troviamo le classi e le interfacce che hanno il compito di rappresentare la struttura dei nostri dati.

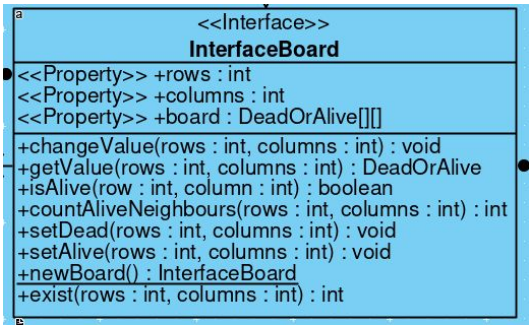
- **DeadOrAlive** : questa enumerazione ci identifica i possibili valori che possono assumere le celle della matrice che ci rappresentano una partita
  DEAD -> cellula morta
  ALIVE -> cellula viva
- **InterfaceBoard** : questa interfaccia viene estesa alla classe che ha il compito di immagazzinare la matrice della partita , questa interfaccia ci permette di imporre dei metodi indispensabili per la gestione della matrice della partita :
  i setter e getter.
  "exist" -> che ci permette di vedere se data la posizione di una cella (date riga e colonna) , questa se appartiene alla matrice si o no.

  "changeValue" -> che permette di cambiare il valore di una cella data la sua posizione nella matrice.

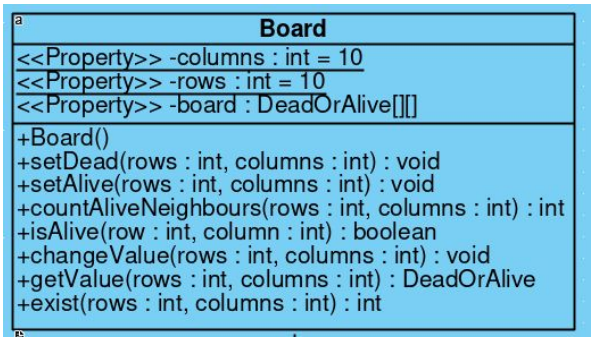
  "countAliveNeighbours" -> che ci permette di calcolare le cellule vive vicine ad un data posizione.
  "setDead" -> che ci permette di impostare una cella come morta (DEAD).

  "setAlive" -> che ci permette di impostare una cella come viva (ALIVE).

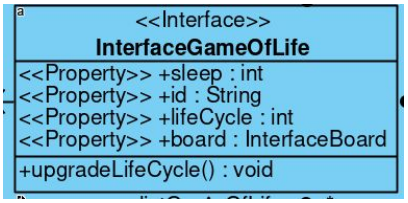
  "isAlive" -> che ci ritorna true se una data posizione contiene una cellula viva altrimenti false.



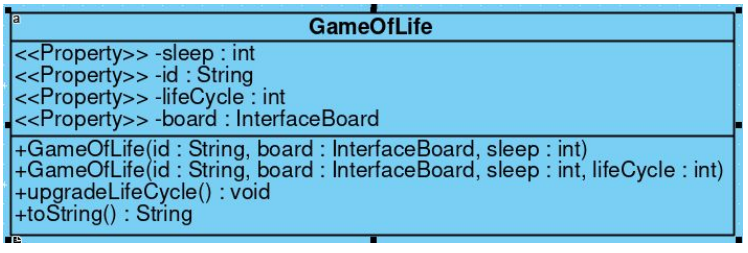
- **Board** : questa classe implementa l'interfaccia InterfaceBoard e quindi anche i metodi detti in precedenza , inoltre qui abbiamo un metodo statico newBoard che ha il compito di creare e ritornare una nuova Board.
  In Board troviamo due constati , che sono il numero delle righe e il numero delle colonne , una matrice di DeadOrAlive che conterrà gli stati di ogni singola cellula.
  Inoltre qui abbiamo il metodo costruttore di questa classe che ha il compito di inizializzare una nuova matrice con tutte le celle impostate su DEAD.



- **InterfaceGameOfLife** : questa interfaccia viene estesa alla classe che ha il compito di strutturare i dati che compongono una partita , qui troveremo i vari getter e setter e il metodo "upgrade LifeCycle" che ha il compito di incrementare di uno la variabile lifeCycle.



- **GameOfLife** : questa classe implementa InterfaceGameOfLife con i relativi metodi detti in precedenza , qui come attributi , troviamo board che e' un InterfaceBoard in modo poter utilizzare qualsiasi classe che estenda questa interfaccia , sleep che e' una variabile che ci identifica i secondi di pausa tra ogni ciclo di vita , id che è il nome della partita e anche il suo identificativo ed lifeCycle che tiene traccia dei cicli di vita che ci sono stati nella seguente partita.
  In questa classe, troviamo due metodi costruttori , uno viene impiegato per quando si crea una nuova partita e gli si passa l'id , board e sleep (non gli si passa lifeCycle per il fatto che in automatico gli si assegna il valore 0 essendo una nuova partita).
  Il secondo metodo costruttore viene utilizzato per quando si carica una partita presente in memoria e quindi gli si passano tutti gli attributi della classe.



- **InterfaceMemoryGame** : questa interfaccia viene implementata dalla classe che ha il compito di di memorizzare dentro se tramite un ArrayList<InterfaceGameOfLife> tutte le partite, qui troviamo :

  i getter e setter dell'ArrayList<InterfaceGameOfLife>

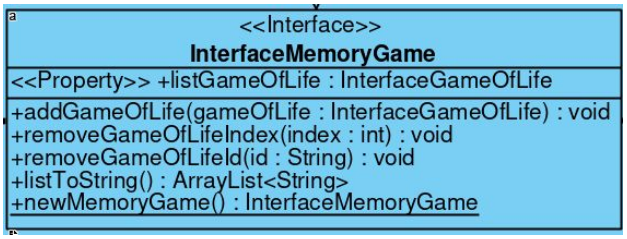
  "addGameOfLife" -> ha il compito di aggiungere un nuovo oggetto che implementa InterfaceGameOfLife nell'ArrayList.

  "removeGameOfLifeIndex" -> ha il compito di rimuovere un elemento nell'ArrayList dato il suo indice.

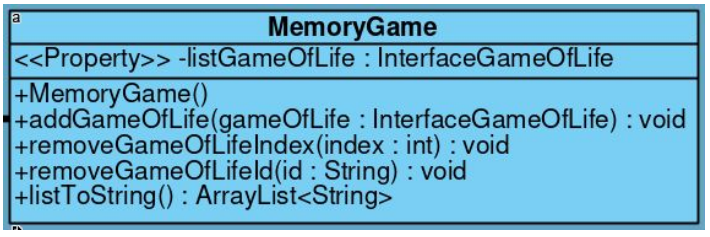
  "removeGameOfLifeId" -> rimuove un elemento nell'ArrayList in base all'id.

  "listToString" -> ha il compito di ritornare un ArrayList<String> dove ad ogni elemento corrisponde una stringa che contiene i dati di una partita.

  "newMemoryGame" -> questo è un metodo statico che ci permette di creare una nuova instanza di MemoryGame.



- **MemoryGame** : classe che implementa InterfaceMemoryGame e che ha il compito di gestire una struttura che andrà memorizzata per salvare le partite inserite.



Inoltre le classi che troviamo in questo package , implementano Serialize , che ci permette di andare a memorizzare gli oggetti direttamente in un file attraverso questo mezzo.

Controller Package

Questo package contiene le classi e le interfacce delle classi che hanno il compito di interagire con il Model e da fare da tramite con la View.

- **InterfaceGameOfLifeController** : questa interfaccia ha il ruolo di gestire ogni azioni che venga eseguita sui dati del Model e di gestire le regole del Gioco.

  "isEmpty" -> metodo per controllare se la lista delle partite e' vuota.

  "addGameOfLife" -> metodo per la creazione di una nuova partita.

  "getListGameOfLife" -> metodo che ritorna la lista delle partite presenti.

  "checkId" -> metodo che controlla se il parametro passato vada bene per il campo id e nel caso che ci siano degli errori come parametro vuoto o presente , ritorna un'eccezione o il parametro.

“checkSleep” -> metodo che controlla se il parametro passato vada bene per il campo sleep e nel caso che ci siano degli errori come parametro vuoto o errori di tipo , ritorna un'eccezione o il valore convertito in int.

“checkColumOrRow” -> controlla se il parametro passato rientri nella dimensione delle colonne o delle righe e ne ritorna il valore convertito in int o un errore.

“checkIndexOfGame” -> controlla se un dato parametro sia un indice della lista delle partite salvate e ritorna il valore inserito -1 o un'eccezione data da un indice errato o un errore di parsing .

“changeValueOfBoard” -> metodo che modifica il valore di una determinata cella di una determinata partita.

“listGameOfLifeToString” -> metodo per ricevere un ArrayList<String> di tutte le partite sotto forma di stringa.

“gameExist” -> metodo che dato un id controlla se esiste una partita caricata con il seguente id e lo ritorna se presente , altrimenti ritorna null se non presente.

“loadGame” -> metodo per caricare la partita selezionata in base all’id.

“nextStep” -> metodo per eseguire un ciclo di vita della paritita passata.

“removeGameOfLifeId” -> metodo per rimuovere una partita dal suo id.

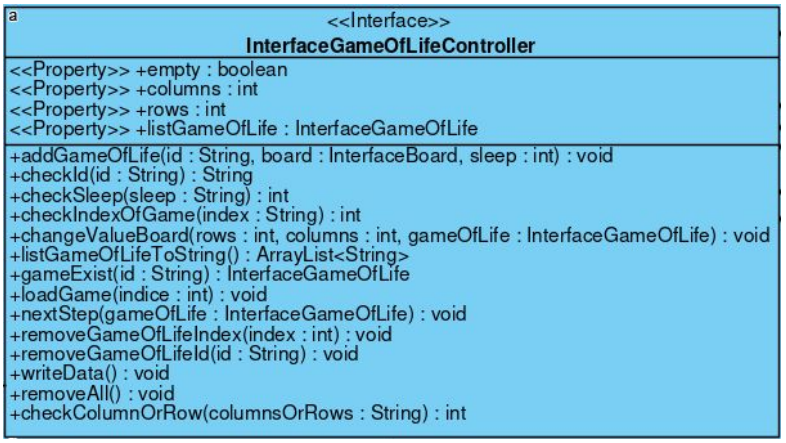
“removeGameOfLifeIndex”-> metodo per rimuovere una partita dal suo indice.

“writeData”->metodo per salvare i dati delle partite in base alla struttura dati passata nel metodo costruttore.

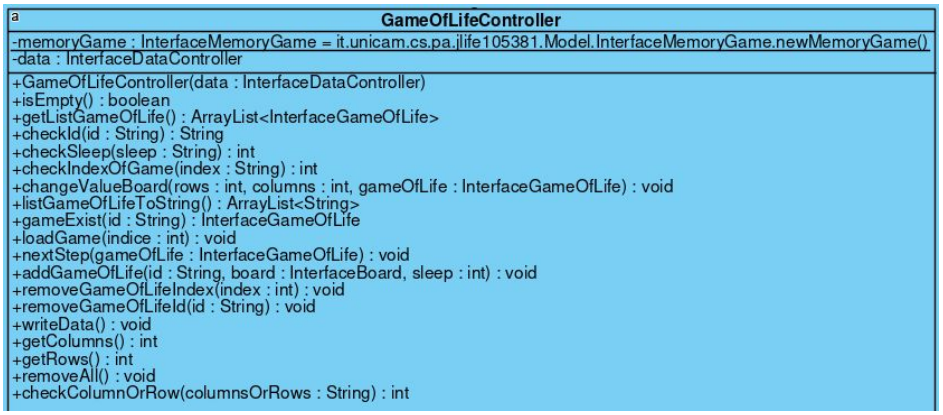
“getColumns” -> metodo che ritorna la dimensione delle colonne della board.

“getRows” -> metodo che ritorna la dimensione delle righe della board.

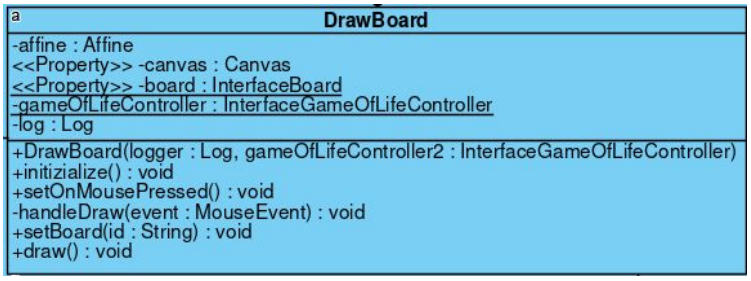
“removeAll” -> metodo per resettare le partite salvate.



- **GameOfLifeController** : classe che implementa InterfaceGameOfLifeController , e che ha il compito di gestire i dati del Model , richiamare l'interfaccia per la memorizzazione/lettura dei dati e il controllare la correttezza dei dati .
  Questa classe implementa un attributo del tipo InterfaceMemoryGame per la l'immagazzinamento dei dati , un attributo del tipo InterfaceDataController per il salvataggio/lettura dei dati.
  Questi attributi vengono inizializzati nel metodo costruttore che dato sia un InterfaceMemoryGame che un InterfaceDataController va a leggere i dati e salvata su memoryGame, nel caso che non ci siano dati salvati,viene inizializzato uno nuovo.



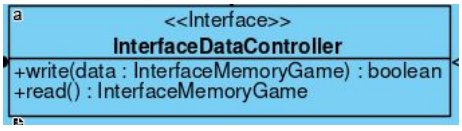
- **DrawBoard** : classe che ha il compito di gestire e creare una tabella dal lato GUI .
  Questa classe ha il compito di creare eventi nel caso che una cella venga premuta e di visualizzare i valori della matrice a video.



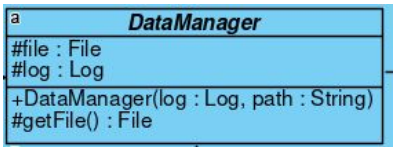
Inoltre nel package Controller abbiamo un altro package "**Data**" che ha il compito di gestire i dati e di memorizzarli sia che con Gson che sia con Serialize, in questo package troviamo :

- **InterfaceDataController** : questa interfaccia ha il compito di memorizzare/leggere i dati da varie strutture.
  "write" -> metodo per la scrittura dei dati.

  "read" -> metodo per la lettura dei dati.

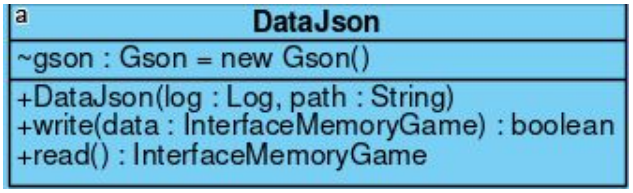


- **DataManager** : si tratta di una classe astratta che ha il compito di aprire un file da un dato path e nel caso che non sia presente di crearne uno nuovo .



- **DataSerialize** : classe che estende DataManager e implementa Serialize e InterfaceDataController, questa classe ha il compito di salvare e leggere i dati tramite l'ausilio di Serialize che dato un oggetto lo salva direttamente sul file e lo legge.

- **DataJson** : classe che estende DataManager e implementa InterfaceDataController, questa classe grazie all'ausilio di Gson, una libreria di Google , possiamo trasformare i nostri Oggetti in json e viceverse.



- **MemoryGameJson** : questa classe ci permette di andare a deserializzare dal json la nostra classe MemoryGame, l'utilizzo di questa classe è dovuto dal fatto che MemoryGame al suo interno contiene altri classi (dipendenza circolare) e quindi dobbiamo andare a specificare il giusto modo per eseguire la Deserializzazione.

View Package

Il nostro package View è suddiviso in due parti per il fatto che possiamo scegliere due tipi di interfacciamento con il programma , quello tramite Console e quello tramite GUI , ma entrambe sono accomunate dall'utilizzo dell'interfaccia InterfaceView.

- **InterfaceView :** questa interfaccia viene utilizzata per far implementare i metodi essenziali per l'utilizzo dell'interfaccia e il corretto funzionamento del programma:

  "start" -> ha il compito di inizializzare / lanciare il programma.

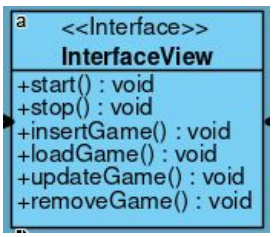
  "stop"-> ha il compito di chiudere i canali aperti e salvare un'ultima volta i dati.

  "insertGame" -> per l'inserimento di una nuova partita.

  "loadGame" -> per il caricamento di una partita.

  "updateGame" -> per la modifica di una partita.

  "removeGame" -> per la rimozione di una partita.

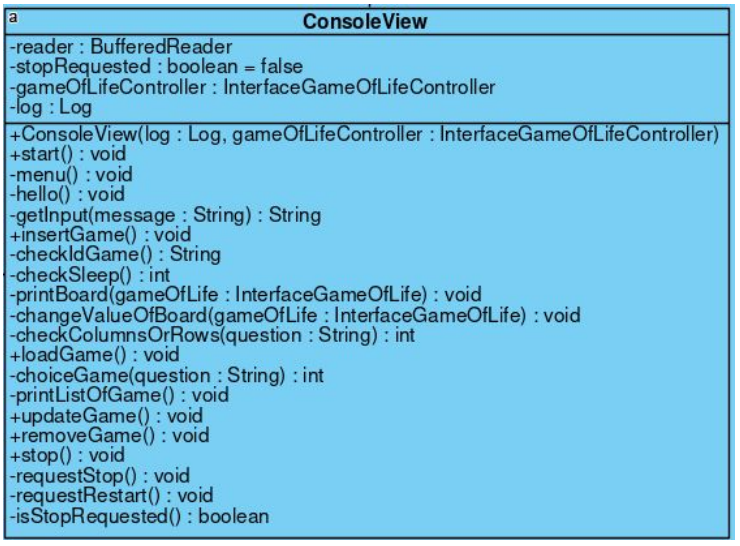


Console Package

- **ConsoleView :** la seguente classe ha il compito di far visualizzare a console un'interfaccia che permetta di far interagire l'utente , questa classe implementa InterfaceView con i suoi rispettivi metodi :

  "insertGame " -> metodo per l'inserimento di una nuova partita che richiede un informazione alla volta , richiamando dei metodi per la gestione di ogni singolo parametro ed in fine eseguendo l'inserimento.

  "loadGame" -> metodo che permette di scegliere la partita da caricare e poi attraverso l'utilizzo di un thread permette di eseguire un ciclo di vita del gioco e stamparlo a video in loop , fino a quando nel thread principale non si riceva un input che permetta di cambiare valore ad una variabile utilizzata per impedire l'uscita dal secondo thread , uno volta uscito dal secondo thread si resetta la variabili in modo da poter richiamare di nuovo questo metodo in un secondo momento.



  "updateGame" -> permette di scegliere la partita da modificare e inserendo le coordinate della cella da modificare gli si cambia di valore fino a quando non viene digitato exit .

  "removeGame" -> permette di scegliere la partita da rimuovere e rimuoverla .

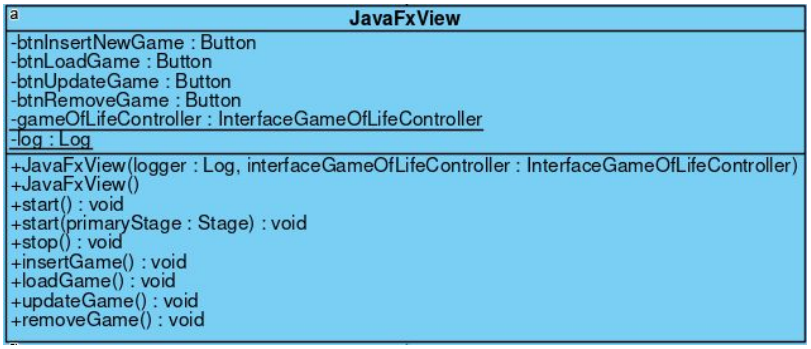
  Questa classe ha come attributo un oggetto che implementa InterfaceGameOfLifeController, che ci permette di eseguire le operazioni di cui avremo bisogna (Controller MVC) .

JavaFx Package

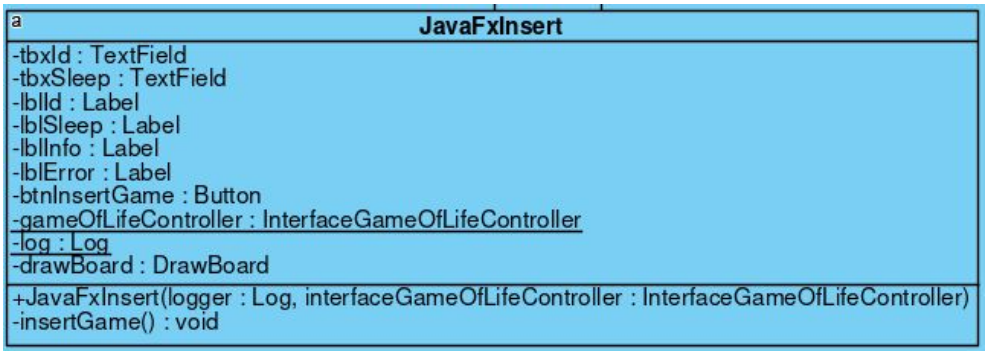
In questo package troviamo le classi utilizzate per interfacciarsi con l'utente tramite GUI.
Per la realizzazione dell'interfaccia grafica si è utilizzato JavaFx , un framework per la realizzazione grafica in java e Scene Builder , un software che semplifica la realizzazione di un'interfaccia creando dei file FXML che ci indicano come l'interfaccia debba essere creata .

- **JavaFxView** : questa è la nostra classe principale per la realizzazione dell'interfaccia grafica , consiste in un menu dove possiamo scegliere l'operazione da eseguire .
  Questa classe implementa InterfaceView, e ciò ci permetterà di andare a implementare i metodi per la realizzazione di ogni comando che dovremo eseguire (insert, load,update,remove) .
  Inoltre JavaFxView farà' partire MenuGameOfLife.fxml che ci rappresenta graficamente come la nostra interfaccia iniziale.
  Per ogni operazione che possiamo fare (insert,load,update,remove) , andremo a creare una nuova interfaccia in APPLICATION MODAL cioè' che finché non si chiude la nuova interfaccia , non potremo eseguire azioni su quella principale dove abbiamo il menu.

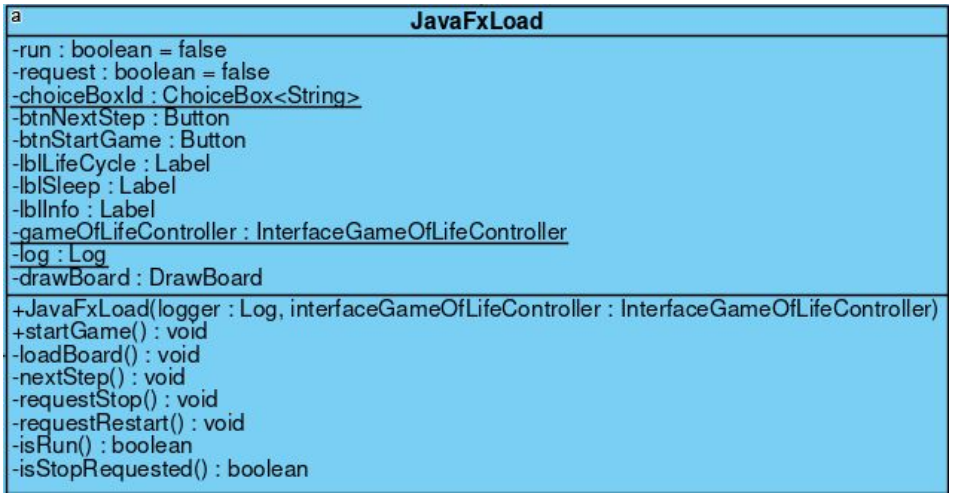
In fine in questa classe troviamo il nostro controller di tipo InterfaceGameOfLifeController che lo passeremo ad ogni nuova interfaccia aperta.



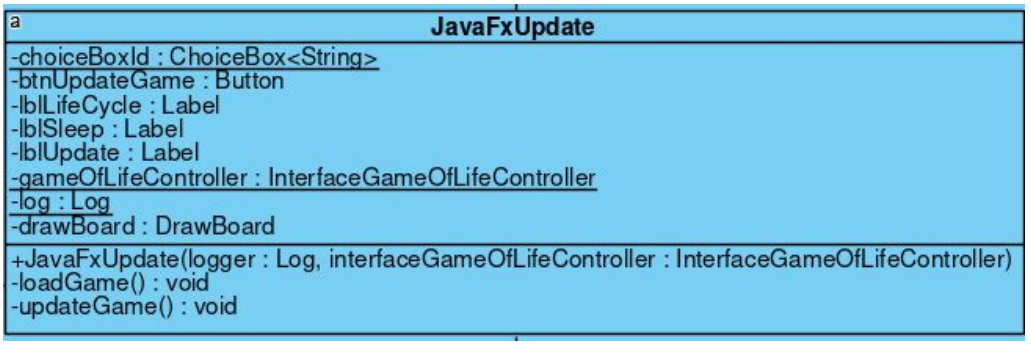
- **JaxaFxInsert** : Questa classe ha lo scopo di andare a creare un'interfaccia per effettuare l'inserimento di una nuova partita , andando a gestire i vari errori di inserimento che si possono effettuare e richiamando la classe DrawBoard per disegnare la matrice.



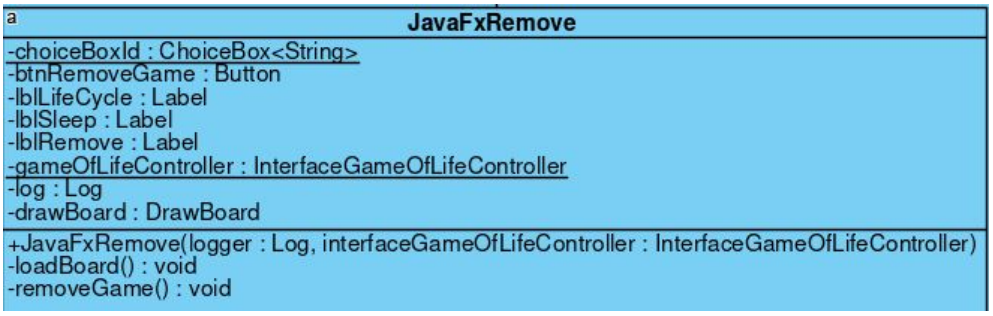
- **JavaFxLoad :** Questa classe permette di andare a caricare la partita che si vuole eseguire, scegliendola dalla lista degli id e caricarla , una volta caricata si possono visualizzare tutte le informazioni della partita e la matrice grazie all'ausilio della classe DrawBoard , una volta caricata si può eseguire un solo step del ciclo di vita , o avviare la partita che andrà a creare un Thread che ad ogni tot di tempo andrà a mostrare il nuovo ciclo di vita della partita (i cambiamenti nel lato grafico sono inseriti in un apposito metodo chiamato Platform.runLater , per il fatto che un thread secondario non può andare a modificare direttamente l'interfaccia grafica in esecuzione) , questo thread sarà interrotto una volta premuto il pulsante stop che richiamerà un metodo synchronized che andrà a modificare una variabile usata nel thread ed questo terminare , ma prima di terminare resetterà questa variabile.



- **JavaFxUpdate** : Questa classe ha il compito di caricare una partita che sarà' scelta da una lista in base al suo id , e una volta caricata si potrà modificare i dati della matrice e salvare .



- **JavaFxRemove** : Questa classe ha il compito di caricare una partita che sarà' scelta da una lista in base al suo id , e una volta caricata si potrà scegliere se eliminarla.



Funzionalità Extra

Nella classe principale App.java , abbiamo la possibilita di andare a scegliere quale applicazione far partire in base al primo parametro passato nel lancio dell'applicazione.

- createGameOfLifeSerializeConsoleView -> utilizzo di Serialize per i dati e Console per la visualizzazione

- createGameOfLifeJsonConsoleView -> utilizzo di json per i dati e Console per visualizzazione

- createGameOfLifeSerializeJavaFx -> utilizzo di Serialize per i dati e GUI per la visualizzazione

- createGameOfLifeJsonJavaFx -> utilizzo di json per i dati e GUI per la visualizzazione

Inoltre andremo a creare un logger che passeremo al nostro controllore per memorizzare i vari errori che si possono avere durante un'esecuzione in modo tale da averli accessibili per effettuare delle future correzioni.

Caricamento -> nella fase di creazione del Controllore si va a leggere i dati da un file in modo da poter caricare partite salvate in precedenza , il fatto di andare ad utilizzare un'interfaccia per la creazione dell'oggetto , ci permette di avere i metodi di cui abbiamo bisogna senza sapere quello che c'è' dietro (DataJson o DataSerialize)

Avvio Partita -> per l'avvio della partita , principalmente ci basiamo sull'utilizzo di un thread che ci permette di eseguire uno step nel ciclo di vita delle cellule ogni tot di tempo (dato dal valore di sleep) fino a quando una certa condizione si avvera ed questa andrà a modificare la variabile che ci impediva di uscire dal thread.

Test

- BoardTest : i test della classe Board , vanno a controllare il corretto funzionamento della creazione di una board , dove tutti gli elementi devono essere vuoti , il cambio di valore di una cella , la conta delle cellule vive vicine a una cella e che se data la posizione di una cella , questa appartiene alla board.

- GameOfLifeTest : testo la creazione di una partita e il suo caricamento , l'incremento del numero di cicli di vita della partita e il metodo toString.

- MemoryGameTest : testo l'inserimento e il get della lista di partite , la rimozione per id e per indice e il metodo che mi ritorna una lista di tutte le partite come stringa.

- GameOfLifeControllerTest : in questa classe vado a controllare il corretto funzionamento di ogni singolo metodo , sia di controllo dei dati (checkId , checkSleep ecc ecc) , se mi ritornano il valore giusto e anche la corretta eccezione in caso di errore , il get e set della lista delle partite , l'inserimento e la rimozione di una partita o di tutte , e il corretto funzionamento del caricamento di una partita con l'esecuzione dei step dei cicli di vita .

Future versioni

Per facilitare la realizzazione di future versione con maggiori funzionalità , si e' fatto utilizzo delle interfacce per la gestione dei dati in modo da garantire una maggiore facilità per le implementazioni di nuovi componenti e la divisione delle classi per responsabilità, inoltre la struttura del programma segue MVC in modo da andare a dividere maggiormente le varie responsabilità.