A4 Design Specification

Michael Ilao, ilaom, 400114462

April 9th 2019

1 Introduction

The design specification describes an implementation of John Conway's Game of Life. The specification uses the model-view-controller design pattern. The Model consists of the GOL_Board and the in_outReader. The in_outReader takes a file of an array of 0's and 1's and converts the information into a string. It also is used to write a string into the same format as the input files. The GOL_Board then uses that string and converts it into a 2D array with matching cells as the input file. The Board then computes the next iteration of the Game of Life, where 0's represent dead cells and 1's represent alive cells. The game can then be viewed by the user using the View Module, viewBoard. The viewBoard module takes a 2D array and prints to the User in the console. The Controller part of the design pattern is taken care of by the test cases, that demonstrate the functionality of the implementation.

2 Critique\Overview

The design specification and implementation demonstrates the following qualities, *Consistency*; The language is clear and concise, the symbols used are all defined within each module and any exported types used throughout the specification are defined in each module used. The order of parameters are always the same throughout the specification and exceptions are used to handle boundary cases. *Essentiality*; the specification is essential throughout as their modules all have unique functions that do not overlap. The I/O module has only one way to write a file and one way read a file. The GOL_Board does violate this quality as toString and getBoardArray do essentialy return the same object however this attributes to understanding the specification and how each module interacts with each other. *Generalilty*; The specification has multiple features to how it can be used as the I/O Module can be used for any program that needs to read and write to a file as it reads in strings and outputs strings and not a 2D array which is only specific to this program. *Minimality*; access routines work together to form a functional design of the game. *Cohesion*; The Components of the game are very closely related as they all need to be used in order to run the game. First the I/O module must be called to read a file. Then

the GOL_Board must be called to create the board and run a simulation. Then the view Module must be called to see the progress of the game. Finally to save the game I/O must be then called again. Finally the specification demonstates $Information Hiding$; as all state variables are private and can only be accessed using getters, though this only returns a copy of the state variable. All access routines are simple in their function though require hidden local functions to fully perform their task.

Input/Output Module

Module

in_outReader

Uses

N/A

Syntax

Exported Constants

N/A

Exported Types

N/A

Exported Access Programs

Routine Name	In	Out	Exceptions
new in_outReader			none
readInput	path : string	data : string	invalid_argument
writeOutput	path : string, data : string		none

Semantics

State Variables

None

State Invariant

None

Access Routine Semantics

new in_outReader():

- exception: none

[Used to create an object to utilize the other functions —MI]

readInput(*path*):

- transition: read data from file assosiated with *path* this data is used to populate a game board table in GOL_Board module.

  This file consists of a text file that is contains a string of 0's and 1's. The file can contain any amount of rows and columns, but each column must be the same size and each row must be the same size. Though column's does not have to be equal to row's. There should be a space between 0's and 1's (ommit commas)

$$\begin{array}{ccccccc} 0/1_{0,0}, & 0/1_{0,1}, & 0/1_{0,2}, & 0/1_{0,3}, & 0/1_{0,4}, & ..., & 0/1_{0,j} \\ 0/1_{1,0}, & 0/1_{1,1}, & 0/1_{1,2}, & 0/1_{1,3}, & 0/1_{1,4}, & ..., & 0/1_{1,j} \\ ..., & ..., & ..., & ..., & ..., & ..., & ..., \\ 0/1_{i,0}, & 0/1_{i,1}, & 0/1_{i,2}, & 0/1_{i,3}, & 0/1_{i,4}, & ..., & 0/1_{i,j} \end{array} \tag{1}$$

- output: Table converted to a string, data.
- exception: (path = NULL $\Rightarrow$ invalid_argument)

[Reads a text file from path and returns a string. Note text file must be in format described —MI]
writeOutput(*path*, *data*):

- transition: Writes a file from *path*, containing *data*
- output: none
- exception: none

[Writes string, data to a file in string, path —MI]

View Module

Module

viewBoard

Uses

N/A

Syntax

Exported Constants

N/A

Exported Types

boardArray = seq of (seq of (e : string))

Exported Access Programs

Routine Name	In	Out	Exceptions
new viewBoard			None
printBoard	game : boardArray		*out_of_range*

State Variables

None

State Invariant

None

Semantics

Enviroment Variables

console: two dimensional sequence of coloured pixels

Access Routine Semantics

new viewBoard():

- exception: none

[Used to utilize print function —MI]

printBoard($game$):

- transition: modify console so ($\forall i : \mathbb{N} | i < |game|. \forall j : \mathbb{N} | j < |game[i]$. game[i][j] is printed on console with " "(empty spaces in between) $\wedge$ j = game[i] $\Rightarrow$ print on console a new line character)
- exception: ($|game| = 0 \vee \exists i : \mathbb{N} | i < |game|. |game[i]| = 0 \Rightarrow out\_of\_range$)

[On the console print index i,j of the 2D array i counts up to size of the array, and j counts up to the size of the array contained in index, i. After printing contents of an index print a space character and if j = to the size of the array print a new line character —MI]

Model Module

Module

GOL_Board

Uses

N/A

Syntax

Exported Constants

N/A

Exported Types

boardArray = seq of (seq of (e : string))

Exported Access Programs

Routine Name	In	Out	Exceptions
new GOL_Board	data : string		None
nextMove			None
toString		str : string	None
getBoardArray		board : boardArray	None

State Variables

game : boardArray
Updatedgame : board Array

State Invariant

None

Semantics

Access Routine Semantics

GOL_Board(*data*)

- transition:
  line : seq of string
  xs = xSize(data)
  ys = ySize(data)

  $\forall i : \mathbb{N} \mid i < |data|. ((data[i] = 1 \Rightarrow line[i\%xs] = data[i]) \vee (data[i] = 0 \Rightarrow line[i\%xs] = data[i])) \wedge (i = xs \Rightarrow (game[i\%ys] = line \Rightarrow line = null))$

  Updatedgame = game
- exception: none

[Creates a new game board with the string, data. updates the *game* state variable by creating adding a sequence of 0's and 1's to an index, i in the *game* array. When the size of a sequence is = size of the row add that sequence to the *game* array. Clear the seqence and move onto the next index in the *game* array —MI]

nextMove()

- transition :
  $\forall i, j : \mathbb{N} \mid i < |game| , j < |game[i]| .$
  $checkStatus(game[i][j], i, j) \Rightarrow Updatedgame[i][j] = 1 \vee$
  $\neg checkStatus(game[i][j], i, j) \Rightarrow Updatedgame[i][j] = 0$
  game = Updatedgame
- exception: none

[Loop through all indicies in the *game* array, call checkStatus, if it returns true than put a 1 in the same index in the *Updatedgame* array. If it returns false put a 0 instead. At the end of the loop set the current *game* to the *Updatedgame* —MI]

toString()

- output: out : string
  $\forall i, j : \mathbb{N} \mid i < |game| , j < |game[i]| . out||game[i][j]||" " \wedge$
  $j = |game[i]| \Rightarrow out||"\backslash n"$
- exception: none

[Build a string of each index in the *game* array with spaces in between, when at the end of a row add a new line character and continue to the next row. —MI]

getBoardArray()

- output : game
- exception : none

Local Functions

xSize: string $\Rightarrow \mathbb{Z}$
xSize(data) $\equiv +$ i : $\mathbb{N}$ | i < $|data|$. (data[i] = 0 $\vee$ data[i] = 1 $\vee$ data[i]) = "\n"
$\Rightarrow$ end
[count the number of 0's and 1's in the first row, once a new line character is reached break from the loop —MI]

ySize: string $\Rightarrow \mathbb{Z}$
ySize(data) $\equiv +$ i : $\mathbb{N}$ | i < $|data|$. (data[i] = "\n")
[count the number of new line characters —MI]

checkstatus: string $\times \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
checkstatus(e, i, j) $\equiv$
alive = + ($\forall$ x,y : $\mathbb{N}$ | x = $\{i-1, i, i+1\}$ $\wedge$ y = $\{j-1, j, j+1\}$:
checkValid(x,y,i,j) $\wedge$ game[x][y] = 1)
output = checkAlive(e, alive)
[for each index check all neighbouring cells if they are valid(checkValid) and equal to one. Then return the checkAlive of the cell and how many neighbouring cells are alive —MI]

checkValid: $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
checkValid(x,y,i,j) $\equiv$
(x > -1 $\wedge$ y > -1) $\wedge$ (x < $|game|$ $\wedge$ y < $|game[0]|$) $\wedge$!(x = i $\wedge$ y = j)
[Checks if the neighbouring cells are valid to check, negative indicies and indicies out of range are ommited and the cell itself are ommited —MI]

checkAlive: string $\times \mathbb{N} \Rightarrow \mathbb{B}$
checkAlive(e, alive) $\equiv$

e\Alive	0	1	2	3	3=<
1	F	F	T	T	F
0	F	F	F	T	F

[Checks if the cell is alive by following the rules of the game, if the cell is alive and has 2 or 3 neighbours it stays alive else it dies. Or if the cell is dead, but had 3 neighbours then it is populated —MI]