Assignment 4, Part 1, Specification

Harshil Modi

April 10, 2019

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game state of a game of life. The game involves a n by m game-board with each location containing a cell. Each cell can be LIVE or DEAD. The following rules apply.

For a cell that is 'LIVE':

- Each cell with one or no neighbours dies, as if by solitude.
- Each cell with four or more neighbours dies, as if by overpopulation.
- Each cell with two or three neighbours survives.

For a cell that is 'DEAD":

- Each cell with three neighbours becomes populated.

Refer to the following website for more information: https://bitstorm.org/gameoflife

Cell Types Module

Module

CellType

Uses

N/A

Syntax

Exported Types

N/A

Exported Types

StateT =$\{LIVE, DEAD\}$
CellT = tuple of (row: $\mathbb{N}$, col: $\mathbb{N}$, s: StateT)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

GameBoard ADT Module

Template Module

GameBoard

Uses

CellType

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
GameBoardT	$\mathbb{N}$, $\mathbb{N}$, Seq of (seq(CellT))	GameBoardT	
set_cell_state	$\mathbb{N}, \mathbb{N}, StateT$		invalid_argument
get_cell_state	$\mathbb{N}, \mathbb{N}$	$StateT$	invalid_argument
get_GameBoard		Seq of (seq(CellT))	
get_total_cells	$StateT$	$\mathbb{N}$	
get_neighbour_state	$\mathbb{N}$, $\mathbb{N}$, $StateT$	$\mathbb{N}$	
iteration	$\mathbb{N}$		

State Variables

Grid: seq of (seq(CellT))
number_rows : $\mathbb{N}$
number_columns: $\mathbb{N}$

State Invariant

None

Assumptions & Design Decisions

- The gameBoardT constructor is called before any other access routine is called on that instance. When the gameBoardT is called the number of rows an columns should match that of grid.

- The grid of the gameBoard is limited to the size of Grid and therefore the corner cells will only have 3 neighbours.
- For better scalability, this module is specified as an Abstract Data Type (ADT) instead of an Abstract Object. This would allow multiple games to be created and tracked at once by a client.

Access Routine Semantics

Grid(row, col, grid):

- transition: Grid, number_rows, number_columns:= grid, row, col
- output out:= self

set_cell_state(row,col, state):

- transition: Grid[row][col].s := state
- exception: (¬ valid_cell_state(row, col, state)) $\implies$ invalid_argument

get_cell_state(row, col):

- out:= Grid[row][col].s
- exception: (¬ valid_cell(row, col)) $\implies$ invalid_argument

get_gameboard():

- out:= Grid
- exception: None

get_total_cells(state):

- out:= $+(\forall i, j : \mathbb{N} | vaild\_cell(i, j) \implies Grid[i][j].s == state : 1)$
- exception: None

get_neighbour_state(row, col, state):

- out:= $+(\forall i, j : \mathbb{N} | vaild\_neighbour\_cell(i, j) \implies Grid[i][j].s == state : 1)$
- exception: None

iteration(int n):

- transition: $(\forall i, j : \mathbb{N} | i, j \in [-1..1] : \neg(i = j) \wedge vaild\_neighbour\_cell(i, j) \implies swap\_state(Grid[i][j].s))$
- exception: None

Local Types

Local Functions

valid_cell: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
valid_cell(row, col) $\equiv$ valid_row(row) $\wedge$ valid_column(col)

valid_cell_state: $\mathbb{N} \times \mathbb{N} \times \mathit{StateT} \rightarrow \mathbb{B}$
valid_cell_state(row, col) $\equiv$ valid_row(row) $\wedge$ valid_column(col) $\wedge$ valid_state(s)

valid_row: $\mathbb{N} \rightarrow \mathbb{B}$
valid_row(row) $\equiv 0 \leq$ row $\wedge$ row $<$ number_rows

valid_col: $\mathbb{N} \rightarrow \mathbb{B}$
valid_col(col) $\equiv 0 \leq$ col $\wedge$ col $<$ number_column

valid_neighnour_cell: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
valid_col(col) $\equiv$ valid_col(j) $\wedge$ valid_row(i)

valid_state: $\mathit{StateT} \rightarrow \mathbb{B}$
valid_state(state) $\equiv$ state = LIVE $\vee$ state = DEAD

swap_state: $\mathbb{N} \times \mathbb{N} \times \mathit{StateT}$
swap_state(i, j, s) $\equiv$

s = DEAD	get_neighbour_state(i, j, LIVE) = 3	set_cell_state(i, j, LIVE)
s = LIVE	get_neighbour_state(i, j, DEAD) > 3	set_cell_state(i, j, DEAD)
	get_neighbour_state(i, j, DEAD) < 2	set_cell_state(i, j, DEAD)

Display Module

Module

Display

Uses

Gameboard, CellType

Syntax

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
write_gamestdnt_state	seq of (seq(CellT))		
display_game_state	seq of (seq(CellT))		
read_game_state		seq of (seq(CellT))	

Semantics

Environment Variables

input_file: File listing initial Gameboard
output_file: File listing GameBoard state

State Variables

input_path : string
output_path: string

State Invariant

None

Assumptions

The input file will match the specification

Assumptions

The input file will match the given specification. The input_file will be present. If the output_file is present, the file will be overwritten and if not a new file with the name will be created.

Access Routine Semantics

read_game_state()

- output: read data from the file input_file associated with the state varaible input_path. Use this data to update and return the state of Seq of (Seq(CellT)).

  The text file has the following format, indicated below. The r_n represent the row index and c_m represents the column index. Each location represents a cell where "X" represents a dead cell and "O" represents a live cell. All Data in the column is separated by a space and each row is separated by a new line character.

$$\begin{array}{ccccc} r_0c_0, & r_0c_1, & r_0c_2, & ..., & r_0c_m \\ r_1c_0, & r_1c_1, & r_1c_2, & ..., & r_1c_m \\ r_2c_0, & r_2c_1, & r_2c_2, & ..., & r_2c_m \\ ..., & ..., & ..., & ..., & ..., \\ r_nc_0, & r_nc_1, & r_nc_2, & ... & r_nc_m \end{array} \tag{1}$$

- exception: none

write_game_state(Grid)

- transition: writes data from Grid to the file output_file where the path of the file can be found in the state variable output_path.

  The text file has the following format, indicated below. The r_n represent the row index and c_m represents the column index. Each location represents a cell where "X" represents a dead cell and "O" represents a live cell. All Data in the column is separated by a space and each row is separated by a new line character.

$$\begin{matrix} r_0c_0, & r_0c_1, & r_0c_2, & ..., & r_0c_m \\ r_1c_0, & r_1c_1, & r_1c_2, & ..., & r_1c_m \\ r_2c_0, & r_2c_1, & r_2c_2, & ..., & r_2c_m \\ ..., & ..., & ..., & ..., & ..., \\ r_nc_0, & r_nc_1, & r_nc_2, & ... & r_nc_m \end{matrix} \tag{2}$$

- exception: none

display_game_state(Grid)

- transition: prints data from Grid to the screen.

  The printed data will be in the following format, indicated below. The r_n represent the row index and c_m represents the column index. Each location represents a cell where "X" represents a dead cell and "O" represents a live cell. All Data in the column is separated by a space and each row is separated by a new line character.

$$\begin{matrix} r_0c_0, & r_0c_1, & r_0c_2, & ..., & r_0c_m \\ r_1c_0, & r_1c_1, & r_1c_2, & ..., & r_1c_m \\ r_2c_0, & r_2c_1, & r_2c_2, & ..., & r_2c_m \\ ..., & ..., & ..., & ..., & ..., \\ r_nc_0, & r_nc_1, & r_nc_2, & ... & r_nc_m \end{matrix} \tag{3}$$

- exception: none

Overview and Critique of my design:

- For my design I decided on using a Structure CellT to represent the cell where each cell contains their location and a enum type StateT which has the state the cell "lIVE" or "DEAD". The reason this was done was for efficiency and for future maintainability. For my design I choose to go with the same rules of the game as game of life however if a cell was a corner cell then the cell only has 3 neighbours instead of 8. The reason for this was for easiness of implementation as time was limited. A seq (seq(CellT)) was used for better visibility and to mimic a 2D array.

- Consistency: Throughout the MIS and software coding portion of this assignment, consistency was greatly considered. This is because being consistent makes the program more usable and user-friendly. An example of this is seem throughout the MIS as only greater then (>), less then (<) and equals symbols are used instead of all three plus greater then or equal too (>=) and Less then or equal too (<=). Throughout the software, consistency was maintained by using Google c++ style guide as convention. This meant that functions were descriptive and setters and getters were used as naming convention (e.g set_cell_state() and get_cell_state). Another convention that was followed was that each word was separated by an underscore rather then capital letters as seen in the example above. Furthermore, constants were all capital letters as seen in Gameboard module and stateT(e.g. DEAD, LIVE). Finally, throughout the Specification the same words were used to maintain consistency rather then use synonyms(e.g row, r, x). The order of input parameters were also kept the same in the order; first: row, second: column, third: stateT.

- Essentiality: Throughout the program, Essentiality was another principle that was greatly taken into consideration. For example, if two functions had similar roles or could do the same thing only one was used. In short, only the necessary functions are included. For example, originally I had set_game_state() and set_cell_live() as my functions in GameBoard. However, after further tweaking the MIS, I was able to see that set_game_state() was able to do everything that set_cell_live() was and as a result set_cell_live() was eliminated. A place that this was violated was by including get_total_neighbours(). This function counts the total number of cells with type StateT (input parameter). This was not used in the current implementation but was included for scalability purposes when in the future we want to check if the state of the game is finished. Currently we were not required to provided the controller module however, this function would be used if we were required too. Another place that Essentiality was violated was for get_cell_state() and get_GameBoard(). by using only get_GameBoard() we could get the state of any cell but in this case get_cell_state() was included for testing purposes since a controller module was not needed.

- Generality: To maintain generality emphasis was put on modules that preformed a more general task. For example, instead of having a module to set_cell_dead() and set_cell_Live() only one module called set_cell_state() where the user can enter the state to set the cell. The same idea was used for get_total_cells() and get_total_neighbour() instead of get_total_live_cells(), get_total_dead_cells(). A more general function iterate() was also implemented. Instead of just iterating once throughout the population, the user can choose how many generations to iterate over. This meant that the user could still iterate over one generation but could also iterate over a set number of generations. Finally, generality was maintained for the size of the game-board. Instead of having an n by n game-board, I decided on having a n by m game-board for a more general shape (rectangle). I wanted to make it even more general by allowing any shape game-board but then the game became much harder to implement and due to time restriction was not implemented.

- Minimality: Minimality was maintained by ensuring that each module only provided one service. This is done by first breaking the modules up into setter and getters. This ensured that one module does not change the state of the game and get the state of the game. For example, get_cell_state() is a getter which retrieves the state of a cell without modifying the state of the game while set_cell_state() modifies the state of the game but does not retrieve the state of the game. This was also taken into consideration for the Display module. Instead of having a write to file module and a print to file module in one method, 2 different methods were used. This was done because one may choose to print to screen but not write to a file at that interval.

- cohesion: The general principle of high cohesion and low coupling was taken into consideration. To ensure High cohesion, methods that are related such as mutators or getters of the state of the game were placed in one module called game-board. On the other hand, methods that had to do with files or printing output to the screen were put in a different module display. The reason I choose to put output to screen in the Display module rather then the Game-Board module is because it is a form of user interaction which is similar to outputting the game-board to a file. Furthermore, Low coupling is done by making display and Game-board as 2 independent modules that do not depend on each other. This ensures that if one of the modules fails, the other module is still working and can function.

- information hiding: For my program, in order to hide the implementation of the logic, I made these functions private functions (e.g. is_valid_state(), is_valid_row()). This reduces the complexity for the user and is great for anticipating for change. In the future, if a better or fast algorithm is available, the user won't know that something was changed. Information hiding was also implemented as it is good practice since if a user knows how something was implemented, it is easier for the user to break/hack it.